CUSPARSE LIBRARY

DU-06709-001_v11.0 | August 2020

TABLE OF CONTENTS

Chapter 1. Introduction....1
1.1. Naming Conventions....1
1.2. Asynchronous Execution....2
1.3. Static Library support....2
Chapter 2. Using the cuSPARSE API....4
2.1. Thread Safety....4
2.2. Scalar Parameters....4
2.3. Parallelism with Streams....4
2.4. Compatibility and Versioning....5
2.5. Optimization Notes....5
Chapter 3. cuSPARSE Indexing and Data Formats....6
3.1. Index Base Format....6
3.1.1. Vector Formats....6
3.1.1.1. Dense Format....6
3.1.1.2. Sparse Format....6
3.2. Matrix Formats....7
3.2.1. Dense Format....7
3.2.2. Coordinate Format (COO)....7
3.2.3. Compressed Sparse Row Format (CSR)....8
3.2.4. Compressed Sparse Column Format (CSC)....9
3.2.5. Block Compressed Sparse Row Format (BSR)....10
3.2.6. Extended BSR Format (BSRX)....12
Chapter 4. cuSPARSE Types Reference....14
4.1. Data types....14
4.2. cusparseStatus_t....14
4.3. cusparseHandle_t....15
4.4. cusparsePointerMode_t....15
4.5. cusparseOperation_t....16
4.6. cusparseAction_t....16
4.7. cusparseDirection_t....16
4.8. cusparseMatDescr_t....17
4.8.1. cusparseDiagType_t....17
4.8.2. cusparseFillMode_t....17
4.8.3. cusparseIndexBase_t....17
4.8.4. cusparseMatrixType_t....17
4.9. cusparseAlgMode_t....18
4.10. cusparseColorInfo_t....18
4.11. cusparseSolvePolicy_t....18
4.12. bsric02Info_t....19
4.13. bsrilu02Info_t....19

4.14. bsrsm2Info_t.. 19
4.15. bsrsv2Info_t.. 19
4.16. csrgemm2Info_t.. 19
4.17. csric02Info_t.. 19
4.18. csrilu02Info_t... 19
4.19. csrsm2Info_t.. 20
4.20. csrsv2Info_t... 20
Chapter 5. cuSPARSE Management Function Reference.. 21
5.1. cusparseCreate()...21
5.2. cusparseDestroy()...21
5.3. cusparseGetErrorName()... 22
5.4. cusparseGetErrorString().. 22
5.5. cusparseGetProperty()... 22
5.6. cusparseGetVersion()..23
5.7. cusparseGetPointerMode()... 23
5.8. cusparseSetPointerMode()..23
5.9. cusparseGetStream()...24
5.10. cusparseSetStream()...24
Chapter 6. cuSPARSE Helper Function Reference... 25
6.1. cusparseCreateColorInfo()..25
6.2. cusparseCreateMatDescr()..25
6.3. cusparseDestroyColorInfo()..26
6.4. cusparseDestroyMatDescr()..26
6.5. cusparseGetMatDiagType()... 26
6.6. cusparseGetMatFillMode()..26
6.7. cusparseGetMatIndexBase()..27
6.8. cusparseGetMatType().. 27
6.9. cusparseSetMatDiagType()..27
6.10. cusparseSetMatFillMode()...28
6.11. cusparseSetMatIndexBase()...28
6.12. cusparseSetMatType()... 28
6.13. cusparseCreateCsrsv2Info()..29
6.14. cusparseDestroyCsrsv2Info()...29
6.15. cusparseCreateCsrsm2Info()..29
6.16. cusparseDestroyCsrsm2Info()...29
6.17. cusparseCreateCsric02Info()...30
6.18. cusparseDestroyCsric02Info()...30
6.19. cusparseCreateCsrilu02Info()...30
6.20. cusparseDestroyCsrilu02Info()...31
6.21. cusparseCreateBsrsv2Info().. 31
6.22. cusparseDestroyBsrsv2Info()... 31
6.23. cusparseCreateBsrsm2Info()... 32
6.24. cusparseDestroyBsrsm2Info()... 32

6.25. cusparseCreateBsric02Info()........32
6.26. cusparseDestroyBsric02Info()........32
6.27. cusparseCreateBsrilu02Info()........33
6.28. cusparseDestroyBsrilu02Info()........33
6.29. cusparseCreateCsrgemm2Info()........33
6.30. cusparseDestroyCsrgemm2Info()........34
6.31. cusparseCreatePruneInfo()........34
6.32. cusparseDestroyPruneInfo()........34
Chapter 7. cuSPARSE Level 1 Function Reference........35
7.1. cusparse<t>axpyi() [DEPRECATED]........36
7.2. cusparse<t>gthr() [DEPRECATED]........37
7.3. cusparse<t>gthrz() [DEPRECATED]........39
7.4. cusparse<t>roti() [DEPRECATED]........40
7.5. cusparse<t>sctr() [DEPRECATED]........42
Chapter 8. cuSPARSE Level 2 Function Reference........44
8.1. cusparse<t>bsrmv()........45
8.2. cusparse<t>bsrxmv()........48
8.3. cusparse<t>bsrsv2_bufferSize()........52
8.4. cusparse<t>bsrsv2_analysis()........55
8.5. cusparse<t>bsrsv2_solve()........58
8.6. cusparseXbsrsv2_zeroPivot()........62
8.7. cusparseCsrmvEx()........64
8.8. cusparse<t>csrsv2_bufferSize()........66
8.9. cusparse<t>csrsv2_analysis()........68
8.10. cusparse<t>csrsv2_solve()........71
8.11. cusparseXcsrsv2_zeroPivot()........75
8.12. cusparse<t>gemvi()........76
Chapter 9. cuSPARSE Level 3 Function Reference........79
9.1. cusparse<t>bsrmm()........80
9.2. cusparse<t>bsrsm2_bufferSize()........84
9.3. cusparse<t>bsrsm2_analysis()........87
9.4. cusparse<t>bsrsm2_solve()........90
9.5. cusparseXbsrsm2_zeroPivot()........93
9.6. cusparse<t>csrsm2_bufferSizeExt()........94
9.7. cusparse<t>csrsm2_analysis()........97
9.8. cusparse<t>csrsm2_solve()........100
9.9. cusparseXcsrsm2_zeroPivot()........103
9.10. cusparse<t>gemmi() [DEPRECATED]........104
Chapter 10. cuSPARSE Extra Function Reference........106
10.1. cusparse<t>csrgeam2()........107
10.2. cusparse<t>csrgemm2() [DEPRECATED]........111
Chapter 11. cuSPARSE Preconditioners Reference........116
11.1. Incomplete Cholesky Factorization: level 0........116

11.1.1. cusparse<t>csric02_bufferSize() 117
11.1.2. cusparse<t>csric02_analysis() 119
11.1.3. cusparse<t>csric02() 121
11.1.4. cusparseXcsric02_zeroPivot() 124
11.1.5. cusparse<t>bsric02_bufferSize() 126
11.1.6. cusparse<t>bsric02_analysis() 128
11.1.7. cusparse<t>bsric02() 131
11.1.8. cusparseXbsric02_zeroPivot() 135
11.2. Incomplete LU Factorization: level 0 135
11.2.1. cusparse<t>csrilu02_numericBoost() 136
11.2.2. cusparse<t>csrilu02_bufferSize() 137
11.2.3. cusparse<t>csrilu02_analysis() 139
11.2.4. cusparse<t>csrilu02() 141
11.2.5. cusparseXcsrilu02_zeroPivot() 144
11.2.6. cusparse<t>bsrilu02_numericBoost() 145
11.2.7. cusparse<t>bsrilu02_bufferSize() 147
11.2.8. cusparse<t>bsrilu02_analysis() 150
11.2.9. cusparse<t>bsrilu02() 153
11.2.10. cusparseXbsrilu02_zeroPivot() 157
11.3. Tridiagonal Solve 157
11.3.1. cusparse<t>gtsv2_buffSizeExt() 158
11.3.2. cusparse<t>gtsv2() 160
11.3.3. cusparse<t>gtsv2_nopivot_bufferSizeExt() 162
11.3.4. cusparse<t>gtsv2_nopivot() 164
11.4. Batched Tridiagonal Solve 165
11.4.1. cusparse<t>gtsv2StridedBatch_bufferSizeExt() 166
11.4.2. cusparse<t>gtsv2StridedBatch() 168
11.4.3. cusparse<t>gtsvInterleavedBatch() 170
11.5. Batched Pentadiagonal Solve 172
11.5.1. cusparse<t>gpsvInterleavedBatch() 173
Chapter 12. cuSPARSE Reorderings Reference 176
12.1. cusparse<t>csrcolor() 177
Chapter 13. cuSPARSE Format Conversion Reference 180
13.1. cusparse<t>bsr2csr() 181
13.2. cusparse<t>gebsr2gebsc() 184
13.3. cusparse<t>gebsr2gebsr() 187
13.4. cusparse<t>gebsr2csr() 191
13.5. cusparse<t>csr2gebsr() 194
13.6. cusparse<t>coo2csr() 197
13.7. cusparse<t>csc2dense() 199
13.8. cusparse<t>csr2bsr() 201
13.9. cusparse<t>csr2coo() 203
13.10. cusparseCsr2cscEx2() 205

13.11. cusparse<t>csr2dense()....207
13.12. cusparse<t>csr2csr_compress()....209
13.13. cusparse<t>dense2csc()....213
13.14. cusparse<t>dense2csr()....215
13.15. cusparse<t>nnz()....217
13.16. cusparseCreateIdentityPermutation()....218
13.17. cusparseXcoosort()....219
13.18. cusparseXcsrsort()....221
13.19. cusparseXcscsort()....223
13.20. cusparseXcsru2csr()....226
13.21. cusparseXpruneDense2csr()....230
13.22. cusparseXpruneCsr2csr()....233
13.23. cusparseXpruneDense2csrPercentage()....236
13.24. cusparseXpruneCsr2csrByPercentage()....239
13.25. cusparse<t>nnz_compress()....242
Chapter 14. cuSPARSE Generic API Reference....244
14.1. Generic Types Reference....244
14.1.1. cudaDataType_t....244
14.1.2. cusparseFormat_t....245
14.1.3. cusparseOrder_t....245
14.1.4. cusparseIndexType_t....246
14.2. Sparse Vector APIs....246
14.2.1. cusparseCreateSpVec()....246
14.2.2. cusparseDestroySpVec()....246
14.2.3. cusparseSpVecGet()....247
14.2.4. cusparseSpVecGetIndexBase()....247
14.2.5. cusparseSpVecGetValues()....248
14.2.6. cusparseSpVecSetValues()....248
14.3. Sparse Matrix APIs....248
14.3.1. cusparseCreateCoo()....248
14.3.2. cusparseCreateCooAoS()....249
14.3.3. cusparseCreateCsr()....250
14.3.4. cusparseDestroySpMat()....250
14.3.5. cusparseCooGet()....251
14.3.6. cusparseCooAosGet()....251
14.3.7. cusparseCsrGet()....252
14.3.8. cusparseCsrSetPointers()....253
14.3.9. cusparseSpMatGetSize()....253
14.3.10. cusparseSpMatGetFormat()....253
14.3.11. cusparseSpMatGetIndexBase()....254
14.3.12. cusparseSpMatGetValues()....254
14.3.13. cusparseSpMatSetValues()....254
14.3.14. cusparseSpMatGetStridedBatch()....255

14.3.15. cusparseSpMatSetStridedBatch() [DEPRECATED].. 255
14.3.16. cusparseCooSetStridedBatch().. 255
14.3.17. cusparseCsrSetStridedBatch().. 256
14.4. Dense Vector APIs.. 256
14.4.1. cusparseCreateDnVec().. 256
14.4.2. cusparseDestroyDnVec()... 257
14.4.3. cusparseDnVecGet().. 257
14.4.4. cusparseDnVecGetValues().. 257
14.4.5. cusparseDnVecSetValues().. 258
14.5. Dense Matrix APIs.. 258
14.5.1. cusparseCreateDnMat()... 258
14.5.2. cusparseDestroyDnMat()... 259
14.5.3. cusparseDnMatGet().. 259
14.5.4. cusparseDnMatGetValues().. 259
14.5.5. cusparseDnSetValues()... 260
14.5.6. cusparseDnMatGetStridedBatch().. 260
14.5.7. cusparseDnMatSetStridedBatch().. 260
14.6. Generic API Functions.. 261
14.6.1. cusparseAxpby()... 261
14.6.2. cusparseGather().. 262
14.6.3. cusparseScatter()... 263
14.6.4. cusparseRot()... 265
14.6.5. cusparseSpVV().. 266
14.6.6. cusparseSpMV().. 268
14.6.7. cusparseSpMM().. 271
14.6.8. cusparseConstrainedGeMM()... 275
14.6.9. cusparseSpGEMM().. 278
Chapter 15. Appendix B: cuSPARSE Fortran Bindings.. 281
15.1. Fortran Application... 283
Chapter 16. Appendix B: Examples of sorting.. 284
16.1. COO Sort.. 284
Chapter 17. Appendix C: Examples of prune.. 286
17.1. Prune Dense to Sparse... 286
17.2. Prune Sparse to Sparse.. 288
17.3. Prune Dense to Sparse by Percentage... 290
17.4. Prune Sparse to Sparse by Percentage.. 292
Chapter 18. Appendix D: Examples of gpsv... 294
18.1. Batched Penta-diagonal Solver... 294
Chapter 19. Appendix E: Examples of csrsm2.. 296
19.1. Forward Triangular Solver... 296
Chapter 20. Appendix F: Acknowledgements.. 298
Chapter 21. Bibliography.. 299

Chapter 1. INTRODUCTION

The cuSPARSE library contains a set of basic linear algebra subroutines used for handling sparse matrices. The library targets matrices with a number of (structural) zero elements which represent > 95% of the total entries.

It is implemented on top of the NVIDIA® CUDA™ runtime (which is part of the CUDA Toolkit) and is designed to be called from C and C++.

The library routines can be classified into four categories:

- Level 1: operations between a vector in sparse format and a vector in dense format
- Level 2: operations between a matrix in sparse format and a vector in dense format
- Level 3: operations between a matrix in sparse format and a set of vectors in dense format (which can also usually be viewed as a dense tall matrix)
- Conversion: operations that allow conversion between different matrix formats, and compression of csr matrices.

The cuSPARSE library allows developers to access the computational resources of the NVIDIA graphics processing unit (GPU), although it does not auto-parallelize across multiple GPUs. The cuSPARSE API assumes that input and output data reside in GPU (device) memory, unless it is explicitly indicated otherwise by the string **DevHostPtr** in a function parameter's name.

It is the responsibility of the developer to allocate memory and to copy data between GPU memory and CPU memory using standard CUDA runtime API routines, such as **cudaMalloc()**, **cudaFree()**, **cudaMemcpy()**, and **cudaMemcpyAsync()**.

1.1. Naming Conventions

The cuSPARSE library functions are available for data types **float**, **double**, **cuComplex**, and **cuDoubleComplex**. The sparse Level 1, Level 2, and Level 3 functions follow this naming convention:

cusparse<**t**>[<**matrix data format**>]<**operation**>[<**output matrix data format**>]

where <**t**> can be **S**, **D**, **C**, **Z**, or **X**, corresponding to the data types **float**, **double**, **cuComplex**, **cuDoubleComplex**, and the generic type, respectively.

The <**matrix data format**> can be **dense**, **coo**, **csr**, or **csc**, corresponding to the dense, coordinate, compressed sparse row, and compressed sparse column formats, respectively.

Finally, the <**operation**> can be **axpyi**, **gthr**, **gthrz**, **roti**, or **sctr**, corresponding to the Level 1 functions; it also can be **mv** or **sv**, corresponding to the Level 2 functions, as well as **mm** or **sm**, corresponding to the Level 3 functions.

All of the functions have the return type **cusparseStatus_t** and are explained in more detail in the chapters that follow.

1.2. Asynchronous Execution

The cuSPARSE library functions are executed asynchronously with respect to the host and may return control to the application on the host before the result is ready. Developers can use the **cudaDeviceSynchronize()** function to ensure that the execution of a particular cuSPARSE library routine has completed.

A developer can also use the **cudaMemcpy()** routine to copy data from the device to the host and vice versa, using the **cudaMemcpyDeviceToHost** and **cudaMemcpyHostToDevice** parameters, respectively. In this case there is no need to add a call to **cudaDeviceSynchronize()** because the call to **cudaMemcpy()** with the above parameters is blocking and completes only when the results are ready on the host.

1.3. Static Library support

Starting with release 6.5, the cuSPARSE Library is also delivered in a static form as libcusparse_static.a on Linux and Mac OSes. The static cuSPARSE library and all others static maths libraries depend on a common thread abstraction layer library called libculibos.a on Linux and Mac and culibos.lib on Windows.

For example, on linux, to compile a small application using cuSPARSE against the dynamic library, the following command can be used:

```
    nvcc myCusparseApp.c  -lcusparse  -o myCusparseApp
```

Whereas to compile against the static cuSPARSE library, the following command has to be used:

```
    nvcc myCusparseApp.c  -lcusparse_static   -lculibos -o myCusparseApp
```

It is also possible to use the native Host C++ compiler. Depending on the Host Operating system, some additional libraries like **pthread** or **dl** might be needed on the linking line. The following command on Linux is suggested :

```
    g++ myCusparseApp.c  -lcusparse_static   -lculibos -lcudart_static -
lpthread -ldl -I <cuda-toolkit-path>/include -L <cuda-toolkit-path>/lib64 -o
 myCusparseApp
```

Note that in the latter case, the library **cuda** is not needed. The CUDA Runtime will try to open explicitly the **cuda** library if needed. In the case of a system which does not have the CUDA driver installed, this allows the application to gracefully manage this issue and potentially run if a CPU-only path is available.

Chapter 2.
USING THE CUSPARSE API

This chapter describes how to use the cuSPARSE library API. It is not a reference for the cuSPARSE API data types and functions; that is provided in subsequent chapters.

2.1. Thread Safety

The library is thread safe and its functions can be called from multiple host threads. However, simultaneous read/writes of the same objects (or of the same handle) are not safe. Hence the handle must be private per thread, i.e., only one handle per thread is safe.

2.2. Scalar Parameters

In the cuSPARSE API, the scalar parameters α and β can be passed by reference on the host or the device.

The few functions that return a scalar result, such as **`nnz()`**, return the resulting value by reference on the host or the device. Even though these functions return immediately, similarly to those that return matrix and vector results, the scalar result is not ready until execution of the routine on the GPU completes. This requires proper synchronization be used when reading the result from the host.

This feature allows the cuSPARSE library functions to execute completely asynchronously using streams, even when α and β are generated by a previous kernel. This situation arises, for example, when the library is used to implement iterative methods for the solution of linear systems and eigenvalue problems [3].

2.3. Parallelism with Streams

If the application performs several small independent computations, or if it makes data transfers in parallel with the computation, CUDA streams can be used to overlap these tasks.

The application can conceptually associate a stream with each task. To achieve the overlap of computation between the tasks, the developer should create CUDA streams using the function **`cudaStreamCreate()`** and set the stream to be used by each individual cuSPARSE library routine by calling **`cusparseSetStream()`** just before calling the actual cuSPARSE routine. Then, computations performed in separate streams would be overlapped automatically on the GPU, when possible. This approach is especially useful when the computation performed by a single task is relatively small and is not enough to fill the GPU with work, or when there is a data transfer that can be performed in parallel with the computation.

When streams are used, we recommend using the new cuSPARSE API with scalar parameters and results passed by reference in the device memory to achieve maximum computational overlap.

Although a developer can create many streams, in practice it is not possible to have more than 16 concurrent kernels executing at the same time.

2.4. Compatibility and Versioning

The cuSPARSE APIs are intended to be backward compatible at the source level with future releases (unless stated otherwise in the release notes of a specific future release). In other words, if a program uses cuSPARSE, it should continue to compile and work correctly with newer versions of cuSPARSE without source code changes. cuSPARSE is not guaranteed to be backward compatible at the binary level. Using different versions of the **`cusparse.h`** header file and the shared library is not supported. Using different versions of cuSPARSE and the CUDA runtime is not supported. The APIs should be backward compatible at the source level for public functions in most cases

2.5. Optimization Notes

Most of the cuSPARSE routines can be optimized by exploiting *CUDA Graphs capture* and *Hardware Memory Compression* features.

More in details, a single cuSPARSE call or a sequence of calls can be captured by a CUDA Graph and executed in a second moment. This minimizes kernels launch overhead and allows the CUDA runtime to optimize the whole workflow. A full example of CUDA graphs capture applied to a cuSPARSE routine can be found in cuSPARSE Library Samples - CUDA Graph.

Secondly, the data types and functionalities involved in cuSPARSE are suitable for *Hardware Memory Compression* available in Ampere GPU devices (compute capability 8.0) or above. The feature allows memory compression for data with enough zero bytes without no loss of information. The device memory must be allocation with the CUDA driver APIs. A full example of Hardware Memory Compression applied to a cuSPARSE routine can be found in cuSPARSE Library Samples - Memory Compression.

Chapter 3. CUSPARSE INDEXING AND DATA FORMATS

The cuSPARSE library supports dense and sparse vector, and dense and sparse matrix formats.

3.1. Index Base Format

The library supports zero- and one-based indexing. The index base is selected through the **`cusparseIndexBase_t`** type, which is passed as a standalone parameter or as a field in the matrix descriptor **`cusparseMatDescr_t`** type.

3.1.1. Vector Formats

This section describes dense and sparse vector formats.

3.1.1.1. Dense Format

Dense vectors are represented with a single data array that is stored linearly in memory, such as the following 7×1 dense vector.

$$\begin{bmatrix} 1.0 & 0.0 & 0.0 & 2.0 & 3.0 & 0.0 & 4.0 \end{bmatrix}$$

(This vector is referenced again in the next section.)

3.1.1.2. Sparse Format

Sparse vectors are represented with two arrays.

- The *data array* has the nonzero values from the equivalent array in dense format.
- The *integer index array* has the positions of the corresponding nonzero values in the equivalent array in dense format.

For example, the dense vector in section 3.2.1 can be stored as a sparse vector with one-based indexing.

$$\begin{bmatrix} 1.0 & 2.0 & 3.0 & 4.0 \end{bmatrix}$$
$$\begin{bmatrix} 1 & 4 & 5 & 7 \end{bmatrix}$$

It can also be stored as a sparse vector with zero-based indexing.

$$\begin{matrix}[1.0 & 2.0 & 3.0 & 4.0] \\ [0 & 3 & 4 & 6\ ]\end{matrix}$$

In each example, the top row is the data array and the bottom row is the index array, and it is assumed that the indices are provided in increasing order and that each index appears only once.

3.2. Matrix Formats

Dense and several sparse formats for matrices are discussed in this section.

3.2.1. Dense Format

The dense matrix **X** is assumed to be stored in column-major format in memory and is represented by the following parameters.

m	(integer)	The number of rows in the matrix.
n	(integer)	The number of columns in the matrix.
ldX	(integer)	The leading dimension of **X**, which must be greater than or equal to **m**. If **ldX** is greater than **m**, then **X** represents a sub-matrix of a larger matrix stored in memory
X	(pointer)	Points to the data array containing the matrix elements. It is assumed that enough storage is allocated for **X** to hold all of the matrix elements and that cuSPARSE library functions may access values outside of the sub-matrix, but will never overwrite them.

For example, **m×n** dense matrix **X** with leading dimension **ldX** can be stored with one-based indexing as shown.

$$\begin{bmatrix} X_{1,1} & X_{1,2} & \cdots & X_{1,n} \\ X_{2,1} & X_{2,2} & \cdots & X_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{m,1} & X_{m,2} & \cdots & X_{m,n} \\ \vdots & \vdots & \ddots & \vdots \\ X_{ldX,1} & X_{ldX,2} & \cdots & X_{ldX,n} \end{bmatrix}$$

Its elements are arranged linearly in memory in the order below.

$$[X_{1,1}\ \ X_{2,1}\ \ \cdots\ \ X_{m,1}\ \ \cdots\ \ X_{ldX,1}\ \ \cdots\ \ X_{1,n}\ \ X_{2,n}\ \ \cdots\ \ X_{m,n}\ \ \cdots\ \ X_{ldX,n}]$$

This format and notation are similar to those used in the NVIDIA CUDA cuBLAS library.

3.2.2. Coordinate Format (COO)

The **m×n** sparse matrix **A** is represented in COO format by the following parameters.

nnz	(integer)	The number of nonzero elements in the matrix.
cooValA	(pointer)	Points to the data array of length **nnz** that holds all nonzero values of **A** in row-major format.
cooRowIndA	(pointer)	Points to the integer array of length **nnz** that contains the row indices of the corresponding elements in array **cooValA**.
cooColIndA	(pointer)	Points to the integer array of length **nnz** that contains the column indices of the corresponding elements in array **cooValA**.

A sparse matrix in COO format is assumed to be stored in row-major format: the index arrays are first sorted by row indices and then within the same row by compressed column indices. It is assumed that each pair of row and column indices appears only once.

For example, consider the following 4×5 matrix **A**.

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 \end{bmatrix}$$

It is stored in COO format with zero-based indexing this way.

$$\begin{aligned} \text{cooValA} &= [1.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 5.0 \quad 7.0 \quad 8.0 \quad 9.0 \quad 6.0] \\ \text{cooRowIndA} &= [0 \quad 0 \quad 1 \quad 1 \quad 2 \quad 2 \quad 2 \quad 3 \quad 3\ ] \\ \text{cooColIndA} &= [0 \quad 1 \quad 1 \quad 2 \quad 0 \quad 3 \quad 4 \quad 2 \quad 4\ ] \end{aligned}$$

In the COO format with one-based indexing, it is stored as shown.

$$\begin{aligned} \text{cooValA} &= [1.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 5.0 \quad 7.0 \quad 8.0 \quad 9.0 \quad 6.0] \\ \text{cooRowIndA} &= [1 \quad 1 \quad 2 \quad 2 \quad 3 \quad 3 \quad 3 \quad 4 \quad 4\ ] \\ \text{cooColIndA} &= [1 \quad 2 \quad 2 \quad 3 \quad 1 \quad 4 \quad 5 \quad 3 \quad 5\ ] \end{aligned}$$

3.2.3. Compressed Sparse Row Format (CSR)

The only way the CSR differs from the COO format is that the array containing the row indices is compressed in CSR format. The **m×n** sparse matrix **A** is represented in CSR format by the following parameters.

nnz	(integer)	The number of nonzero elements in the matrix.
csrValA	(pointer)	Points to the data array of length **nnz** that holds all nonzero values of A in row-major format.
csrRowPtrA	(pointer)	Points to the integer array of length **m+1** that holds indices into the arrays **csrColIndA** and **csrValA**. The first **m** entries of this array contain the indices of the first nonzero element in the **i**th row for **i=i,...,m**, while the last entry contains **nnz+csrRowPtrA(0)**. In general, **csrRowPtrA(0)** is **0** or **1** for zero- and one-based indexing, respectively.
csrColIndA	(pointer)	Points to the integer array of length **nnz** that contains the column indices of the corresponding elements in array **csrValA**.

Sparse matrices in CSR format are assumed to be stored in row-major CSR format, in other words, the index arrays are first sorted by row indices and then within the same row by column indices. It is assumed that each pair of row and column indices appears only once.

Consider again the 4×5matrix**A**.

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 \end{bmatrix}$$

It is stored in CSR format with zero-based indexing as shown.

$$\begin{aligned} \text{csrValA} &= [1.0 \;\; 4.0 \;\; 2.0 \;\; 3.0 \;\; 5.0 \;\; 7.0 \;\; 8.0 \;\; 9.0 \;\; 6.0] \\ \text{csrRowPtrA} &= [0 \;\; 2 \;\; 4 \;\; 7 \;\; 9 \;] \\ \text{csrColIndA} &= [0 \;\; 1 \;\; 1 \;\; 2 \;\; 0 \;\; 3 \;\; 4 \;\; 2 \;\; 4 \;] \end{aligned}$$

This is how it is stored in CSR format with one-based indexing.

$$\begin{aligned} \text{csrValA} &= [1.0 \;\; 4.0 \;\; 2.0 \;\; 3.0 \;\; 5.0 \;\; 7.0 \;\; 8.0 \;\; 9.0 \;\; 6.0] \\ \text{csrRowPtrA} &= [1 \;\; 3 \;\; 5 \;\; 8 \;\; 10 \;] \\ \text{csrColIndA} &= [1 \;\; 2 \;\; 2 \;\; 3 \;\; 1 \;\; 4 \;\; 5 \;\; 3 \;\; 5 \;] \end{aligned}$$

3.2.4. Compressed Sparse Column Format (CSC)

The CSC format is different from the COO format in two ways: the matrix is stored in column-major format, and the array containing the column indices is compressed in CSC format. The **m×n** matrix **A** is represented in CSC format by the following parameters.

nnz	(integer)	The number of nonzero elements in the matrix.
cscValA	(pointer)	Points to the data array of length **nnz** that holds all nonzero values of **A** in column-major format.
cscRowIndA	(pointer)	Points to the integer array of length **nnz** that contains the row indices of the corresponding elements in array **cscValA**.
cscColPtrA	(pointer)	Points to the integer array of length **n+1** that holds indices into the arrays **cscRowIndA** and **cscValA**. The first **n** entries of this array contain the indices of the first nonzero element in the **i**th row for **i=i,...,n**, while the last entry contains **nnz+cscColPtrA(0)**. In general, **cscColPtrA(0)** is **0** or **1** for zero- and one-based indexing, respectively.

The matrix **A** in CSR format has exactly the same memory layout as its transpose in CSC format (and vice versa).

For example, consider once again the 4×5 matrix **A**.

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 \end{bmatrix}$$

It is stored in CSC format with zero-based indexing this way.

$$\begin{aligned} \text{cscValA} &= [1.0 \quad 5.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 9.0 \quad 7.0 \quad 8.0 \quad 6.0] \\ \text{cscRowIndA} &= [0 \quad 2 \quad 0 \quad 1 \quad 1 \quad 3 \quad 2 \quad 2 \quad 3 \ ] \\ \text{cscColPtrA} &= [0 \quad 2 \quad 4 \quad 6 \quad 7 \quad 9 \ ] \end{aligned}$$

In CSC format with one-based indexing, this is how it is stored.

$$\begin{aligned} \text{cscValA} &= [1.0 \quad 5.0 \quad 4.0 \quad 2.0 \quad 3.0 \quad 9.0 \quad 7.0 \quad 8.0 \quad 6.0] \\ \text{cscRowIndA} &= [1 \quad 3 \quad 1 \quad 2 \quad 2 \quad 4 \quad 3 \quad 3 \quad 4 \ ] \\ \text{cscColPtrA} &= [1 \quad 3 \quad 5 \quad 7 \quad 8 \quad 10 \ ] \end{aligned}$$

Each pair of row and column indices appears only once.

3.2.5. Block Compressed Sparse Row Format (BSR)

The only difference between the CSR and BSR formats is the format of the storage element. The former stores primitive data types (**single**, **double**, **cuComplex**, and **cuDoubleComplex**) whereas the latter stores a two-dimensional square block of primitive data types. The dimension of the square block is $blockDim$. The **m×n** sparse matrix **A** is equivalent to a block sparse matrix A_b with $mb = \frac{m + blockDim - 1}{blockDim}$ block rows and $nb = \frac{n + blockDim - 1}{blockDim}$ block columns. If m or n is not multiple of $blockDim$, then zeros are filled into A_b.

A is represented in BSR format by the following parameters.

blockDim	(integer)	Block dimension of matrix **A**.
mb	(integer)	The number of block rows of **A**.
nb	(integer)	The number of block columns of **A**.
nnzb	(integer)	The number of nonzero blocks in the matrix.
bsrValA	(pointer)	Points to the data array of length $nnzb * blockDim^2$ that holds all elements of nonzero blocks of **A**. The block elements are stored in either column-major order or row-major order.
bsrRowPtrA	(pointer)	Points to the integer array of length **mb+1** that holds indices into the arrays **bsrColIndA** and **bsrValA**. The first **mb** entries of this array contain the indices of the first nonzero block in the **i**th block row for **i=1,...,mb**, while the last entry contains **nnzb+bsrRowPtrA(0)**. In general, **bsrRowPtrA(0)** is **0** or **1** for zero- and one-based indexing, respectively.
bsrColIndA	(pointer)	Points to the integer array of length **nnzb** that contains the column indices of the corresponding blocks in array **bsrValA**.

As with CSR format, (row, column) indices of BSR are stored in row-major order. The index arrays are first sorted by row indices and then within the same row by column indices.

For example, consider again the **4×5** matrix **A**.

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 \end{bmatrix}$$

If $blockDim$ is equal to 2, then mb is 2, nb is 3, and matrix **A** is split into **2×3** block matrix A_b. The dimension of A_b is **4×6**, slightly bigger than matrix A, so zeros are filled in the last column of A_b. The element-wise view of A_b is this.

$$\begin{bmatrix} 1.0 & 4.0 & 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 2.0 & 3.0 & 0.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 0.0 & 7.0 & 8.0 & 0.0 \\ 0.0 & 0.0 & 9.0 & 0.0 & 6.0 & 0.0 \end{bmatrix}$$

Based on zero-based indexing, the block-wise view of A_b can be represented as follows.

$$A_b = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

The basic element of BSR is a nonzero A_{ij} block, one that contains at least one nonzero element of **A**. Five of six blocks are nonzero in A_b.

$$A_{00} = \begin{bmatrix} 1 & 4 \\ 0 & 2 \end{bmatrix}, A_{01} = \begin{bmatrix} 0 & 0 \\ 3 & 0 \end{bmatrix}, A_{10} = \begin{bmatrix} 5 & 0 \\ 0 & 0 \end{bmatrix}, A_{11} = \begin{bmatrix} 0 & 7 \\ 9 & 0 \end{bmatrix}, A_{12} = \begin{bmatrix} 8 & 0 \\ 6 & 0 \end{bmatrix}$$

BSR format only stores the information of nonzero blocks, including block indices (*i*, *j*) and values A_{ij}. Also row indices are compressed in CSR format.

$$\begin{aligned} \text{bsrValA} &= [A_{00} \;\; A_{01} \;\; A_{10} \;\; A_{11} \;\; A_{12}] \\ \text{bsrRowPtrA} &= [0 \quad 2 \quad 5] \\ \text{bsrColIndA} &= [0 \quad 1 \quad 0 \quad 1 \quad 2] \end{aligned}$$

There are two ways to arrange the data element of block A_{ij}: row-major order and column-major order. Under column-major order, the physical storage of **bsrValA** is this.

$$bsrValA = [1 \;\; 0 \;\; 4 \;\; 2 \;\; | 0 \;\; 3 \;\; 0 \;\; 0 \;\; | 5 \;\; 0 \;\; 0 \;\; 0 \;\; | 0 \;\; 9 \;\; 7 \;\; 0 \;\; | 8 \;\; 6 \;\; 0 \;\; 0 \;\;]$$

Under row-major order, the physical storage of **bsrValA** is this.

$$bsrValA = [1 \;\; 4 \;\; 0 \;\; 2 \;\; | 0 \;\; 0 \;\; 3 \;\; 0 \;\; | 5 \;\; 0 \;\; 0 \;\; 0 \;\; | 0 \;\; 7 \;\; 9 \;\; 0 \;\; | 8 \;\; 0 \;\; 6 \;\; 0 \;\;]$$

Similarly, in BSR format with one-based indexing and column-major order, **A** can be represented by the following.

$$A_b = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

$$bsrValA = [1 \;\; 0 \;\; 4 \;\; 2 \;\; | 0 \;\; 3 \;\; 0 \;\; 0 \;\; | 5 \;\; 0 \;\; 0 \;\; 0 \;\; | 0 \;\; 9 \;\; 7 \;\; 0 \;\; | 8 \;\; 6 \;\; 0 \;\; 0 \;\;]$$

$$\text{bsrRowPtrA} = [1 \quad 3 \quad 6]$$
$$\text{bsrColIndA} = [1 \quad 2 \quad 1 \quad 2 \quad 3]$$

The general BSR format has two parameters, **rowBlockDim** and **colBlockDim**. **rowBlockDim** is number of rows within a block and **colBlockDim** is number of columns within a block. If **rowBlockDim=colBlockDim**, general BSR format is the same as BSR format. If **rowBlockDim=colBlockDim=1**, general BSR format is the same as CSR format. The conversion routine gebsr2gebsr is used to do conversion among CSR, BSR and general BSR.

In the cuSPARSE Library, the storage format of blocks in BSR format can be column-major or row-major, independently of the base index. However, if the developer uses BSR format from the Math Kernel Library (MKL) and wants to directly interface with the cuSPARSE Library, then **cusparseDirection_t CUSPARSE_DIRECTION_COLUMN** should be used if the base index is one; otherwise, **cusparseDirection_t CUSPARSE_DIRECTION_ROW** should be used.

3.2.6. Extended BSR Format (BSRX)

BSRX is the same as the BSR format, but the array **bsrRowPtrA** is separated into two parts. The first nonzero block of each row is still specified by the array **bsrRowPtrA**, which is the same as in BSR, but the position next to the last nonzero block of each row is specified by the array **bsrEndPtrA**. Briefly, BSRX format is simply like a 4-vector variant of BSR format.

Matrix **A** is represented in BSRX format by the following parameters.

blockDim	(integer)	Block dimension of matrix **A**.
mb	(integer)	The number of block rows of **A**.
nb	(integer)	The number of block columns of **A**.
nnzb	(integer)	number of nonzero blocks in the matrix **A**.
bsrValA	(pointer)	Points to the data array of length $nnzb * blockDim^2$ that holds all the elements of the nonzero blocks of **A**. The block elements are stored in either column-major order or row-major order.
bsrRowPtrA	(pointer)	Points to the integer array of length **mb** that holds indices into the arrays **bsrColIndA** and **bsrValA**; **bsrRowPtrA(i)** is the position of the first nonzero block of the **i**th block row in **bsrColIndA** and **bsrValA**.
bsrEndPtrA	(pointer)	Points to the integer array of length **mb** that holds indices into the arrays **bsrColIndA** and **bsrValA**; **bsrRowPtrA(i)** is the position next to the last nonzero block of the **i**th block row in **bsrColIndA** and **bsrValA**.
bsrColIndA	(pointer)	Points to the integer array of length **nnzb** that contains the column indices of the corresponding blocks in array **bsrValA**.

A simple conversion between BSR and BSRX can be done as follows. Suppose the developer has a **2×3** block sparse matrix A_b represented as shown.

$$A_b = \begin{bmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{bmatrix}$$

Assume it has this BSR format.

$$
\begin{aligned}
\text{bsrValA of BSR} &= [A_{00} \; A_{01} \; A_{10} \; A_{11} \; A_{12}] \\
\text{bsrRowPtrA of BSR} &= [0 \quad 2 \quad 5] \\
\text{bsrColIndA of BSR} &= [0 \quad 1 \quad 0 \quad 1 \quad 2]
\end{aligned}
$$

The **bsrRowPtrA** of the BSRX format is simply the first two elements of the **bsrRowPtrA** BSR format. The **bsrEndPtrA** of BSRX format is the last two elements of the **bsrRowPtrA** of BSR format.

$$
\begin{aligned}
\text{bsrRowPtrA of BSRX} &= [0 \quad 2] \\
\text{bsrEndPtrA of BSRX} &= [2 \quad 5]
\end{aligned}
$$

The advantage of the BSRX format is that the developer can specify a submatrix in the original BSR format by modifying **bsrRowPtrA** and **bsrEndPtrA** while keeping **bsrColIndA** and **bsrValA** unchanged.

For example, to create another block matrix $\tilde{A} = \begin{bmatrix} O & O & O \\ O & A_{11} & O \end{bmatrix}$ that is slightly different from A, the developer can keep **bsrColIndA** and **bsrValA**, but reconstruct $\tilde{A}$ by properly setting of **bsrRowPtrA** and **bsrEndPtrA**. The following 4-vector characterizes $\tilde{A}$.

$$
\begin{aligned}
\text{bsrValA of } \tilde{A} &= [A_{00} \; A_{01} \; A_{10} \; A_{11} \; A_{12}] \\
\text{bsrColIndA of } \tilde{A} &= [0 \quad 1 \quad 0 \quad 1 \quad 2] \\
\text{bsrRowPtrA of } \tilde{A} &= [0 \quad 3] \\
\text{bsrEndPtrA of } \tilde{A} &= [0 \quad 4]
\end{aligned}
$$

Chapter 4. CUSPARSE TYPES REFERENCE

4.1. Data types

The **float**, **double**, **cuComplex**, and **cuDoubleComplex** data types are supported. The first two are standard C data types, while the last two are exported from **cuComplex.h**.

4.2. cusparseStatus_t

This data type represents the status returned by the library functions and it can have the following values

Value	Description
CUSPARSE_STATUS_SUCCESS	The operation completed successfully
CUSPARSE_STATUS_NOT_INITIALI	The cuSPARSE library was not initialized. This is usually caused by the lack of a prior call, an error in the CUDA Runtime API called by the cuSPARSE routine, or an error in the hardware setup **To correct:** call **cusparseCreate()** prior to the function call; and check that the hardware, an appropriate version of the driver, and the cuSPARSE library are correctly installed The error also applies to generic APIs (Generic APIs reference) for indicating a matrix/vector descriptor not initialized
CUSPARSE_STATUS_ALLOC_FAILI	Resource allocation failed inside the cuSPARSE library. This is usually caused by a device memory allocation (**cudaMalloc()**) or by a host memory allocation failure **To correct:** prior to the function call, deallocate previously allocated memory as much as possible
CUSPARSE_STATUS_INVALID_VAL	An unsupported value or parameter was passed to the function (a negative vector size, for example)

Value	Description
	To correct: ensure that all the parameters being passed have valid values
CUSPARSE_STATUS_ARCH_MISMAT	The function requires a feature absent from the device architecture **To correct:** compile and run the application on a device with appropriate compute capability
CUSPARSE_STATUS_EXECUTION_FAI	The GPU program failed to execute. This is often caused by a launch failure of the kernel on the GPU, which can be caused by multiple reasons **To correct:** check that the hardware, an appropriate version of the driver, and the cuSPARSE library are correctly installed
CUSPARSE_STATUS_INTERNAL_ERI	An internal cuSPARSE operation failed **To correct:** check that the hardware, an appropriate version of the driver, and the cuSPARSE library are correctly installed. Also, check that the memory passed as a parameter to the routine is not being deallocated prior to the routine completion
CUSPARSE_STATUS_MATRIX_TYPE_NOT_S	The matrix type is not supported by this function. This is usually caused by passing an invalid matrix descriptor to the function **To correct:** check that the fields in **`cusparseMatDescr_t descrA`** were set correctly
CUSPARSE_STATUS_NOT_SUPPORT	The operation or data type combination is currently not supported by the function
CUSPARSE_STATUS_INSUFFICIENT_RES	The resources for the computation, such as GPU global or shared memory, are not sufficient to complete the operation. The error can also indicate that the current computation mode (e.g. bit size of sparse matrix indices) does not allow to handle the given input

4.3. cusparseHandle_t

This is a pointer type to an opaque cuSPARSE context, which the user must initialize by calling prior to calling **`cusparseCreate()`** any other library function. The handle created and returned by **`cusparseCreate()`** must be passed to every cuSPARSE function.

4.4. cusparsePointerMode_t

This type indicates whether the scalar values are passed by reference on the host or device. It is important to point out that if several scalar values are passed by reference in the function call, all of them will conform to the same single pointer mode. The pointer mode can be set and retrieved using **cusparseSetPointerMode()** and **cusparseGetPointerMode()** routines, respectively.

Value	Meaning
CUSPARSE_POINTER_MODE_HOST	the scalars are passed by reference on the host.
CUSPARSE_POINTER_MODE_DEVICE	the scalars are passed by reference on the device.

4.5. cusparseOperation_t

This type indicates which operations need to be performed with the sparse matrix.

Value	Meaning
CUSPARSE_OPERATION_NON_TRANSPOSE	the non-transpose operation is selected.
CUSPARSE_OPERATION_TRANSPOSE	the transpose operation is selected.
CUSPARSE_OPERATION_CONJUGATE_TRANSPOSE	the conjugate transpose operation is selected.

4.6. cusparseAction_t

This type indicates whether the operation is performed only on indices or on data and indices.

Value	Meaning
CUSPARSE_ACTION_SYMBOLIC	the operation is performed only on indices.
CUSPARSE_ACTION_NUMERIC	the operation is performed on data and indices.

4.7. cusparseDirection_t

This type indicates whether the elements of a dense matrix should be parsed by rows or by columns (assuming column-major storage in memory of the dense matrix) in function cusparse[S|D|C|Z]nnz. Besides storage format of blocks in BSR format is also controlled by this type.

Value	Meaning
CUSPARSE_DIRECTION_ROW	the matrix should be parsed by rows.
CUSPARSE_DIRECTION_COLUMN	the matrix should be parsed by columns.

4.8. cusparseMatDescr_t

This structure is used to describe the shape and properties of a matrix.

```
typedef struct {
    cusparseMatrixType_t MatrixType;
    cusparseFillMode_t FillMode;
    cusparseDiagType_t DiagType;
    cusparseIndexBase_t IndexBase;
} cusparseMatDescr_t;
```

4.8.1. cusparseDiagType_t

This type indicates if the matrix diagonal entries are unity. The diagonal elements are always assumed to be present, but if **CUSPARSE_DIAG_TYPE_UNIT** is passed to an API routine, then the routine assumes that all diagonal entries are unity and will not read or modify those entries. Note that in this case the routine assumes the diagonal entries are equal to one, regardless of what those entries are actually set to in memory.

Value	Meaning
CUSPARSE_DIAG_TYPE_NON_UNIT	the matrix diagonal has non-unit elements.
CUSPARSE_DIAG_TYPE_UNIT	the matrix diagonal has unit elements.

4.8.2. cusparseFillMode_t

This type indicates if the lower or upper part of a matrix is stored in sparse storage.

Value	Meaning
CUSPARSE_FILL_MODE_LOWER	the lower triangular part is stored.
CUSPARSE_FILL_MODE_UPPER	the upper triangular part is stored.

4.8.3. cusparseIndexBase_t

This type indicates if the base of the matrix indices is zero or one.

Value	Meaning
CUSPARSE_INDEX_BASE_ZERO	the base index is zero.
CUSPARSE_INDEX_BASE_ONE	the base index is one.

4.8.4. cusparseMatrixType_t

This type indicates the type of matrix stored in sparse storage. Notice that for symmetric, Hermitian and triangular matrices only their lower or upper part is assumed to be stored.

The whole idea of matrix type and fill mode is to keep minimum storage for symmetric/Hermitian matrix, and also to take advantage of symmetric property on SpMV

(Sparse Matrix Vector multiplication). To compute **y=A*x** when **A** is symmetric and only lower triangular part is stored, two steps are needed. First step is to compute **y=(L+D)*x** and second step is to compute **y=L^T*x + y**. Given the fact that the transpose operation **y=L^T*x** is 10x slower than non-transpose version **y=L*x**, the symmetric property does not show up any performance gain. It is better for the user to extend the symmetric matrix to a general matrix and apply **y=A*x** with matrix type **CUSPARSE_MATRIX_TYPE_GENERAL**.

In general, SpMV, preconditioners (incomplete Cholesky or incomplete LU) and triangular solver are combined together in iterative solvers, for example PCG and GMRES. If the user always uses general matrix (instead of symmetric matrix), there is no need to support other than general matrix in preconditioners. Therefore the new routines, **[bsr|csr]sv2** (triangular solver), **[bsr|csr]ilu02** (incomplete LU) and **[bsr|csr]ic02** (incomplete Cholesky), only support matrix type **CUSPARSE_MATRIX_TYPE_GENERAL**.

Value	Meaning
CUSPARSE_MATRIX_TYPE_GENERAL	the matrix is general.
CUSPARSE_MATRIX_TYPE_SYMMETRIC	the matrix is symmetric.
CUSPARSE_MATRIX_TYPE_HERMITIAN	the matrix is Hermitian.
CUSPARSE_MATRIX_TYPE_TRIANGULAR	the matrix is triangular.

4.9. cusparseAlgMode_t

This is type for algorithm parameter to cusparseCsrmvEx() and cusparseCsrmvEx_bufferSize() functions.

Value	Meaning
CUSPARSE_ALG_MERGE_PATH	Use load-balancing algorithm that suits better for irregular nonzero-patterns.

4.10. cusparseColorInfo_t

This is a pointer type to an opaque structure holding the information used in **csrcolor()**.

4.11. cusparseSolvePolicy_t

This type indicates whether level information is generated and used in **csrsv2, csric02, csrilu02, bsrsv2, bsric02 and bsrilu02**.

Value	Meaning
CUSPARSE_SOLVE_POLICY_NO_LEVEL	no level information is generated and used.

Value	Meaning
`CUSPARSE_SOLVE_POLICY_USE_LEVEL`	generate and use level information.

4.12. bsric02Info_t

This is a pointer type to an opaque structure holding the information used in **`bsric02_bufferSize()`**, **`bsric02_analysis()`**, and **`bsric02()`**.

4.13. bsrilu02Info_t

This is a pointer type to an opaque structure holding the information used in **`bsrilu02_bufferSize()`**, **`bsrilu02_analysis()`**, and **`bsrilu02()`**.

4.14. bsrsm2Info_t

This is a pointer type to an opaque structure holding the information used in **`bsrsm2_bufferSize()`**, **`bsrsm2_analysis()`**, and **`bsrsm2_solve()`**.

4.15. bsrsv2Info_t

This is a pointer type to an opaque structure holding the information used in **`bsrsv2_bufferSize()`**, **`bsrsv2_analysis()`**, and **`bsrsv2_solve()`**.

4.16. csrgemm2Info_t

This is a pointer type to an opaque structure holding the information used in **`csrgemm2_bufferSizeExt()`**, and **`csrgemm2()`**.

4.17. csric02Info_t

This is a pointer type to an opaque structure holding the information used in **`csric02_bufferSize()`**, **`csric02_analysis()`**, and **`csric02()`**.

4.18. csrilu02Info_t

This is a pointer type to an opaque structure holding the information used in **`csrilu02_bufferSize()`**, **`csrilu02_analysis()`**, and **`csrilu02()`**.

4.19. csrsm2Info_t

This is a pointer type to an opaque structure holding the information used in **`csrsm2_bufferSize()`**, **`csrsm2_analysis()`**, and **`csrsm2_solve()`**.

4.20. csrsv2Info_t

This is a pointer type to an opaque structure holding the information used in **`csrsv2_bufferSize()`**, **`csrsv2_analysis()`**, and **`csrsv2_solve()`**.

Chapter 5. CUSPARSE MANAGEMENT FUNCTION REFERENCE

The cuSPARSE functions for managing the library are described in this section.

5.1. cusparseCreate()

```
cusparseStatus_t
cusparseCreate(cusparseHandle_t *handle)
```

This function initializes the cuSPARSE library and creates a handle on the cuSPARSE context. It must be called before any other cuSPARSE API function is invoked. It allocates hardware resources necessary for accessing the GPU.

Param.	In/out	Meaning
`handle`	IN	The pointer to the handle to the cuSPARSE context

See `cusparseStatus_t` for the description of the return status

5.2. cusparseDestroy()

```
cusparseStatus_t
cusparseDestroy(cusparseHandle_t handle)
```

This function releases CPU-side resources used by the cuSPARSE library. The release of GPU-side resources may be deferred until the application shuts down.

Param.	In/out	Meaning
`handle`	IN	The handle to the cuSPARSE context

See `cusparseStatus_t` for the description of the return status

5.3. cusparseGetErrorName()

```
const char*
cusparseGetErrorString(cusparseStatus_t status)
```

The function returns the string representation of an error code enum name. If the error code is not recognized, "unrecognized error code" is returned.

Param.	In/out	Meaning
status	IN	Error code to convert to string
const char*	OUT	Pointer to a NULL-terminated string

5.4. cusparseGetErrorString()

```
const char*
cusparseGetErrorString(cusparseStatus_t status)
```

Returns the description string for an error code. If the error code is not recognized, "unrecognized error code" is returned.

Param.	In/out	Meaning
status	IN	Error code to convert to string
const char*	OUT	Pointer to a NULL-terminated string

5.5. cusparseGetProperty()

```
cusparseStatus_t
cusparseGetProperty(libraryPropertyType type,
                    int*                value)
```

The function returns the value of the requested property. Refer to **libraryPropertyType** for supported types.

Param.	In/out	Meaning
type	IN	Requested property
value	OUT	Value of the requested property

libraryPropertyType (defined in **library_types.h**):

Value	Meaning
MAJOR_VERSION	Enumerator to query the major version
MINOR_VERSION	Enumerator to query the minor version
PATCH_LEVEL	Number to identify the patch level

See cusparseStatus_t for the description of the return status

5.6. cusparseGetVersion()

```
cusparseStatus_t
cusparseGetVersion(cusparseHandle_t handle,
                   int*             version)
```

This function returns the version number of the cuSPARSE library.

Param.	In/out	Meaning
handle	IN	cuSPARSE handle
version	OUT	The version number of the library

See cusparseStatus_t for the description of the return status

5.7. cusparseGetPointerMode()

```
cusparseStatus_t
cusparseGetPointerMode(cusparseHandlet handle,
                       cusparsePointerMode_t *mode)
```

This function obtains the pointer mode used by the cuSPARSE library. Please see the section on the **cusparsePointerMode_t** type for more details.

Param.	In/out	Meaning
handle	IN	The handle to the cuSPARSE context
mode	OUT	One of the enumerated pointer mode types

See cusparseStatus_t for the description of the return status

5.8. cusparseSetPointerMode()

```
cusparseStatus_t
cusparseSetPointerMode(cusparseHandle_t handle,
                       cusparsePointerMode_t mode)
```

This function sets the pointer mode used by the cuSPARSE library. The *default* is for the values to be passed by reference on the host. Please see the section on the **cublasPointerMode_t** type for more details.

Param.	In/out	Meaning
handle	IN	The handle to the cuSPARSE context
mode	IN	One of the enumerated pointer mode types

See cusparseStatus_t for the description of the return status

5.9. cusparseGetStream()

```
cusparseStatus_t
cusparseGetStream(cusparseHandle_t handle, cudaStream_t *streamId)
```

This function gets the cuSPARSE library stream, which is being used to to execute all calls to the cuSPARSE library functions. If the cuSPARSE library stream is not set, all kernels use the default NULL stream.

Param.	In/out	Meaning
handle	IN	The handle to the cuSPARSE context
streamId	OUT	The stream used by the library

See `cusparseStatus_t` for the description of the return status

5.10. cusparseSetStream()

```
cusparseStatus_t
cusparseSetStream(cusparseHandle_t handle, cudaStream_t streamId)
```

This function sets the stream to be used by the cuSPARSE library to execute its routines.

Param.	In/out	Meaning
handle	IN	The handle to the cuSPARSE context
streamId	IN	The stream to be used by the library

See `cusparseStatus_t` for the description of the return status

Chapter 6. CUSPARSE HELPER FUNCTION REFERENCE

The cuSPARSE helper functions are described in this section.

6.1. cusparseCreateColorInfo()

```
cusparseStatus_t
cusparseCreateColorInfo(cusparseColorInfo_t* info)
```

This function creates and initializes the **cusparseColorInfo_t** structure to *default* values.

Input

info	the pointer to the **cusparseColorInfo_t** structure

See cusparseStatus_t for the description of the return status

6.2. cusparseCreateMatDescr()

```
cusparseStatus_t
cusparseCreateMatDescr(cusparseMatDescr_t *descrA)
```

This function initializes the matrix descriptor. It sets the fields **MatrixType** and **IndexBase** to the *default* values **CUSPARSE_MATRIX_TYPE_GENERAL** and **CUSPARSE_INDEX_BASE_ZERO** , respectively, while leaving other fields uninitialized.

Input

descrA	the pointer to the matrix descriptor.

See cusparseStatus_t for the description of the return status

6.3. cusparseDestroyColorInfo()

```
cusparseStatus_t
cusparseDestroyColorInfo(cusparseColorInfo_t info)
```

This function destroys and releases any memory required by the structure.

Input

info	the pointer to the structure of **csrcolor()**

See cusparseStatus_t for the description of the return status

6.4. cusparseDestroyMatDescr()

```
cusparseStatus_t
cusparseDestroyMatDescr(cusparseMatDescr_t descrA)
```

This function releases the memory allocated for the matrix descriptor.

Input

descrA	the matrix descriptor.

See cusparseStatus_t for the description of the return status

6.5. cusparseGetMatDiagType()

```
cusparseDiagType_t
cusparseGetMatDiagType(const cusparseMatDescr_t descrA)
```

This function returns the **DiagType** field of the matrix descriptor **descrA**.

Input

descrA	the matrix descriptor.

Returned

	One of the enumerated diagType types.

6.6. cusparseGetMatFillMode()

```
cusparseFillMode_t
cusparseGetMatFillMode(const cusparseMatDescr_t descrA)
```

This function returns the **FillMode** field of the matrix descriptor **descrA**.

Input

descrA	the matrix descriptor.

Returned

	One of the enumerated fillMode types.

6.7. cusparseGetMatIndexBase()

```
cusparseIndexBase_t
cusparseGetMatIndexBase(const cusparseMatDescr_t descrA)
```

This function returns the **IndexBase** field of the matrix descriptor **descrA**.

Input

descrA	the matrix descriptor.

Returned

	One of the enumerated indexBase types.

6.8. cusparseGetMatType()

```
cusparseMatrixType_t
cusparseGetMatType(const cusparseMatDescr_t descrA)
```

This function returns the **MatrixType** field of the matrix descriptor **descrA**.

Input

descrA	the matrix descriptor.

Returned

	One of the enumerated matrix types.

6.9. cusparseSetMatDiagType()

```
cusparseStatus_t
cusparseSetMatDiagType(cusparseMatDescr_t descrA,
                       cusparseDiagType_t diagType)
```

This function sets the **DiagType** field of the matrix descriptor **descrA**.

Input

diagType	One of the enumerated diagType types.

Output

descrA	the matrix descriptor.

See `cusparseStatus_t` for the description of the return status

6.10. cusparseSetMatFillMode()

```
cusparseStatus_t
cusparseSetMatFillMode(cusparseMatDescr_t descrA,
                       cusparseFillMode_t fillMode)
```

This function sets the **FillMode** field of the matrix descriptor **descrA**.

Input

fillMode	One of the enumerated fillMode types.

Output

descrA	the matrix descriptor.

See cusparseStatus_t for the description of the return status

6.11. cusparseSetMatIndexBase()

```
cusparseStatus_t
cusparseSetMatIndexBase(cusparseMatDescr_t descrA,
                        cusparseIndexBase_t base)
```

This function sets the **IndexBase** field of the matrix descriptor **descrA**.

Input

base	One of the enumerated indexBase types.

Output

descrA	the matrix descriptor.

See cusparseStatus_t for the description of the return status

6.12. cusparseSetMatType()

```
cusparseStatus_t
cusparseSetMatType(cusparseMatDescr_t descrA, cusparseMatrixType_t type)
```

This function sets the **MatrixType** field of the matrix descriptor **descrA**.

Input

type	One of the enumerated matrix types.

Output

descrA	the matrix descriptor.

See cusparseStatus_t for the description of the return status

6.13. cusparseCreateCsrsv2Info()

```
cusparseStatus_t
cusparseCreateCsrsv2Info(csrsv2Info_t *info);
```

This function creates and initializes the solve and analysis structure of csrsv2 to *default* values.

Input

info	the pointer to the solve and analysis structure of csrsv2.

See cusparseStatus_t for the description of the return status

6.14. cusparseDestroyCsrsv2Info()

```
cusparseStatus_t
cusparseDestroyCsrsv2Info(csrsv2Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

info	the solve **(csrsv2_solve)** and analysis **(csrsv2_analysis)** structure.

See cusparseStatus_t for the description of the return status

6.15. cusparseCreateCsrsm2Info()

```
cusparseStatus_t
cusparseCreateCsrsm2Info(csrsm2Info_t *info);
```

This function creates and initializes the solve and analysis structure of csrsm2 to *default* values.

Input

info	the pointer to the solve and analysis structure of csrsm2.

See cusparseStatus_t for the description of the return status

6.16. cusparseDestroyCsrsm2Info()

```
cusparseStatus_t
cusparseDestroyCsrsm2Info(csrsm2Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

`info`	the solve `(csrsm2_solve)` and analysis `(csrsm2_analysis)` structure.

See `cusparseStatus_t` for the description of the return status

6.17. cusparseCreateCsric02Info()

```
cusparseStatus_t
cusparseCreateCsric02Info(csric02Info_t *info);
```

This function creates and initializes the solve and analysis structure of incomplete Cholesky to *default* values.

Input

`info`	the pointer to the solve and analysis structure of incomplete Cholesky.

See `cusparseStatus_t` for the description of the return status

6.18. cusparseDestroyCsric02Info()

```
cusparseStatus_t
cusparseDestroyCsric02Info(csric02Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

`info`	the solve `(csric02_solve)` and analysis `(csric02_analysis)` structure.

See `cusparseStatus_t` for the description of the return status

6.19. cusparseCreateCsrilu02Info()

```
cusparseStatus_t
cusparseCreateCsrilu02Info(csrilu02Info_t *info);
```

This function creates and initializes the solve and analysis structure of incomplete LU to *default* values.

Input

`info`	the pointer to the solve and analysis structure of incomplete LU.

See cusparseStatus_t for the description of the return status

6.20. cusparseDestroyCsrilu02Info()

```
cusparseStatus_t
cusparseDestroyCsrilu02Info(csrilu02Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

info	the solve **(csrilu02_solve)** and analysis **(csrilu02_analysis)** structure.

See cusparseStatus_t for the description of the return status

6.21. cusparseCreateBsrsv2Info()

```
cusparseStatus_t
cusparseCreateBsrsv2Info(bsrsv2Info_t *info);
```

This function creates and initializes the solve and analysis structure of bsrsv2 to *default* values.

Input

info	the pointer to the solve and analysis structure of bsrsv2.

See cusparseStatus_t for the description of the return status

6.22. cusparseDestroyBsrsv2Info()

```
cusparseStatus_t
cusparseDestroyBsrsv2Info(bsrsv2Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

info	the solve **(bsrsv2_solve)** and analysis **(bsrsv2_analysis)** structure.

See cusparseStatus_t for the description of the return status

6.23. cusparseCreateBsrsm2Info()

```
cusparseStatus_t
cusparseCreateBsrsm2Info(bsrsm2Info_t *info);
```

This function creates and initializes the solve and analysis structure of bsrsm2 to *default* values.

Input

`info`	the pointer to the solve and analysis structure of bsrsm2.

See `cusparseStatus_t` for the description of the return status

6.24. cusparseDestroyBsrsm2Info()

```
cusparseStatus_t
cusparseDestroyBsrsm2Info(bsrsm2Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

`info`	the solve `(bsrsm2_solve)` and analysis `(bsrsm2_analysis)` structure.

See `cusparseStatus_t` for the description of the return status

6.25. cusparseCreateBsric02Info()

```
cusparseStatus_t
cusparseCreateBsric02Info(bsric02Info_t *info);
```

This function creates and initializes the solve and analysis structure of block incomplete Cholesky to *default* values.

Input

`info`	the pointer to the solve and analysis structure of block incomplete Cholesky.

See `cusparseStatus_t` for the description of the return status

6.26. cusparseDestroyBsric02Info()

```
cusparseStatus_t
cusparseDestroyBsric02Info(bsric02Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

`info`	the solve `(bsric02_solve)` and analysis `(bsric02_analysis)` structure.

See `cusparseStatus_t` for the description of the return status

6.27. cusparseCreateBsrilu02Info()

```
cusparseStatus_t
cusparseCreateBsrilu02Info(bsrilu02Info_t *info);
```

This function creates and initializes the solve and analysis structure of block incomplete LU to *default* values.

Input

`info`	the pointer to the solve and analysis structure of block incomplete LU.

See `cusparseStatus_t` for the description of the return status

6.28. cusparseDestroyBsrilu02Info()

```
cusparseStatus_t
cusparseDestroyBsrilu02Info(bsrilu02Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

`info`	the solve `(bsrilu02_solve)` and analysis `(bsrilu02_analysis)` structure.

See `cusparseStatus_t` for the description of the return status

6.29. cusparseCreateCsrgemm2Info()

```
cusparseStatus_t
cusparseCreateCsrgemm2Info(csrgemm2Info_t *info);
```

This function creates and initializes analysis structure of general sparse matrix-matrix multiplication.

Input

`info`	the pointer to the analysis structure of general sparse matrix-matrix multiplication.

See cusparseStatus_t for the description of the return status

6.30. cusparseDestroyCsrgemm2Info()

```
cusparseStatus_t
cusparseDestroyCsrgemm2Info(csrgemm2Info_t info);
```

This function destroys and releases any memory required by the structure.

Input

info	opaque structure of csrgemm2.

See cusparseStatus_t for the description of the return status

6.31. cusparseCreatePruneInfo()

```
cusparseStatus_t
cusparseCreatePruneInfo(pruneInfo_t *info);
```

This function creates and initializes structure of **prune** to *default* values.

Input

info	the pointer to the structure of **prune**.

See cusparseStatus_t for the description of the return status

6.32. cusparseDestroyPruneInfo()

```
cusparseStatus_t
cusparseDestroyPruneInfo(pruneInfo_t info);
```

This function destroys and releases any memory required by the structure.

Input

info	the structure of **prune**.

See cusparseStatus_t for the description of the return status

Chapter 7. CUSPARSE LEVEL 1 FUNCTION REFERENCE

This chapter describes sparse linear algebra functions that perform operations between dense and sparse vectors.

7.1. cusparse<t>axpyi() [DEPRECATED]

[[DEPRECATED]] use **cusparseAxpby()** instead. *The routine will be removed in the next major release*

```
usparseStatus_t
cusparseSaxpyi(cusparseHandle_t    handle,
               int                 nnz,
               const float*        alpha,
               const float*        xVal,
               const int*          xInd,
               float*              y,
               cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseDaxpyi(cusparseHandle_t    handle,
               int                 nnz,
               const double*       alpha,
               const double*       xVal,
               const int*          xInd,
               double*             y,
               cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseCaxpyi(cusparseHandle_t    handle,
               int                 nnz,
               const cuComplex*    alpha,
               const cuComplex*    xVal,
               const int*          xInd,
               cuComplex*          y,
               cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseZaxpyi(cusparseHandle_t       handle,
               int                    nnz,
               const cuDoubleComplex* alpha,
               const cuDoubleComplex* xVal,
               const int*             xInd,
               cuDoubleComplex*       y,
               cusparseIndexBase_t    idxBase)
```

This function multiplies the vector **x** in sparse format by the constant α and adds the result to the vector **y** in dense format. This operation can be written as

$$y = y + \alpha * x$$

In other words,

```
for i=0 to nnz-1
    y[xInd[i]-idxBase] = y[xInd[i]-idxBase] + alpha*xVal[i]
```

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
alpha	<type> scalar used for multiplication.
xVal	<type> vector with **nnz** nonzero values of vector **x**.
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
y	<type> vector in dense format.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

y	<type> updated vector in dense format (that is unchanged if **nnz == 0**).

See `cusparseStatus_t` for the description of the return status

7.2. cusparse<t>gthr() [DEPRECATED]

[[DEPRECATED]] use **`cusparseGather()`** instead. *The routine will be removed in the next major release*

```
cusparseStatus_t
cusparseSgthr(cusparseHandle_t    handle,
              int                 nnz,
              const float*        y,
              float*              xVal,
              const int*          xInd,
              cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseDgthr(cusparseHandle_t    handle,
              int                 nnz,
              const double*       y,
              double*             xVal,
              const int*          xInd,
              cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseCgthr(cusparseHandle_t    handle,
              int                 nnz,
              const cuComplex*    y,
              cuComplex*          xVal,
              const int*          xInd,
              cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseZgthr(cusparseHandle_t        handle,
              int                     nnz,
              const cuDoubleComplex*  y,
              cuDoubleComplex*        xVal,
              const int*              xInd,
              cusparseIndexBase_t     idxBase)
```

This function gathers the elements of the vector **y** listed in the index array **xInd** into the data array **xVal**.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
y	<type> vector in dense format (of **size≥max(xInd)-idxBase+1**).
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

xVal	<type> vector with **nnz** nonzero values that were gathered from vector **y** (that is unchanged if **nnz == 0**).

See `cusparseStatus_t` for the description of the return status

7.3. cusparse<t>gthrz() [DEPRECATED]

[[DEPRECATED]] use **`cusparseGather()`** instead. *The routine will be removed in the next major release*

```
cusparseStatus_t
cusparseSgthrz(cusparseHandle_t    handle,
               int                 nnz,
               float*              y,
               float*              xVal,
               const int*          xInd,
               cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseDgthrz(cusparseHandle_t    handle,
               int                 nnz,
               double*             y,
               double*             xVal,
               const int*          xInd,
               cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseCgthrz(cusparseHandle_t    handle,
               int                 nnz,
               cuComplex*          y,
               cuComplex*          xVal,
               const int*          xInd,
               cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseZgthrz(cusparseHandle_t    handle,
               int                 nnz,
               cuDoubleComplex*    y,
               cuDoubleComplex*    xVal,
               const int*          xInd,
               cusparseIndexBase_t idxBase)
```

This function gathers the elements of the vector **`y`** listed in the index array **`xInd`** into the data array **`xVal`**. Also, it zeros out the gathered elements in the vector **`y`**.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`nnz`	number of elements in vector **`x`**.
`y`	<type> vector in dense format (of **`size≥max(xInd)-idxBase+1`**).
`xInd`	integer vector with **`nnz`** indices of the nonzero values of vector **`x`**.

idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

xVal	<type> vector with **nnz** nonzero values that were gathered from vector **y** (that is unchanged if **nnz == 0**).
y	<type> vector in dense format with elements indexed by **xInd** set to zero (it is unchanged if **nnz == 0**).

See `cusparseStatus_t` for the description of the return status

7.4. cusparse<t>roti() [DEPRECATED]

[[DEPRECATED]] use **`cusparseRot()`** instead. *The routine will be removed in the next major release*

```
cusparseStatus_t
cusparseSroti(cusparseHandle_t    handle,
              int                 nnz,
              float*              xVal,
              const int*          xInd,
              float*              y,
              const float*        c,
              const float*        s,
              cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseDroti(cusparseHandle_t    handle,
              int                 nnz,
              double*             xVal,
              const int*          xInd,
              double*             y,
              const double*       c,
              const double*       s,
              cusparseIndexBase_t idxBase)
```

This function applies the Givens rotation matrix

$$G = \begin{pmatrix} c & s \\ -s & c \end{pmatrix}$$

to sparse **x** and dense **y** vectors. In other words,

```
for i=0 to nnz-1
    y[xInd[i]-idxBase] = c * y[xInd[i]-idxBase] - s*xVal[i]
    x[i]               = c * xVal[i]            + s * y[xInd[i]-idxBase]
```

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
nnz	number of elements in vector **x**.
xVal	<type> vector with **nnz** nonzero values of vector **x**.
xInd	integer vector with **nnz** indices of the nonzero values of vector **x**.
y	<type> vector in dense format.
c	cosine element of the rotation matrix.
s	sine element of the rotation matrix.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

xVal	<type> updated vector in sparse format (that is unchanged if **nnz** == 0).
y	<type> updated vector in dense format (that is unchanged if **nnz** == 0).

See cusparseStatus_t for the description of the return status

7.5. cusparse<t>sctr() [DEPRECATED]

[[DEPRECATED]] use `cusparseScatter()` instead. *The routine will be removed in the next major release*

```
cusparseStatus_t
cusparseSsctr(cusparseHandle_t    handle,
              int                 nnz,
              const float*        xVal,
              const int*          xInd,
              float*              y,
              cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseDsctr(cusparseHandle_t    handle,
              int                 nnz,
              const double*       xVal,
              const int*          xInd,
              double*             y,
              cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseCsctr(cusparseHandle_t    handle,
              int                 nnz,
              const cuComplex*    xVal,
              const int*          xInd,
              cuComplex*          y,
              cusparseIndexBase_t idxBase)

cusparseStatus_t
cusparseZsctr(cusparseHandle_t       handle,
              int                    nnz,
              const cuDoubleComplex* xVal,
              const int*             xInd,
              cuDoubleComplex*       y,
              cusparseIndexBase_t    idxBase)
```

This function scatters the elements of the vector **`x`** in sparse format into the vector **`y`** in dense format. It modifies only the elements of **`y`** whose indices are listed in the array **`xInd`**.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`nnz`	number of elements in vector **`x`**.
`xVal`	<type> vector with **`nnz`** nonzero values of vector **`x`**.
`xInd`	integer vector with **`nnz`** indices of the nonzero values of vector **`x`**.

y	<type> dense vector (of **size≥max(xInd)-idxBase+1**).
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

y	<type> vector with **nnz** nonzero values that were scattered from vector **x** (that is unchanged if **nnz == 0**).

See `cusparseStatus_t` for the description of the return status

Chapter 8. CUSPARSE LEVEL 2 FUNCTION REFERENCE

This chapter describes the sparse linear algebra functions that perform operations between sparse matrices and dense vectors.

In particular, the solution of sparse triangular linear systems is implemented in two phases. First, during the analysis phase, the sparse triangular matrix is analyzed to determine the dependencies between its elements by calling the appropriate **`csrsv2_analysis()`** function. The analysis is specific to the sparsity pattern of the given matrix and to the selected **`cusparseOperation_t`** type. The information from the analysis phase is stored in the parameter of type **`csrsv2Info_t`** that has been initialized previously with a call to **`cusparseCreateCsrsv2Info()`**.

Second, during the solve phase, the given sparse triangular linear system is solved using the information stored in the **`csrsv2Info_t`** parameter by calling the appropriate **`csrsv2_solve()`** function. The solve phase may be performed multiple times with different right-hand sides, while the analysis phase needs to be performed only once. This is especially useful when a sparse triangular linear system must be solved for a set of different right-hand sides one at a time, while its coefficient matrix remains the same.

Finally, once all the solves have completed, the opaque data structure pointed to by the **`csrsv2Info_t`** parameter can be released by calling **`cusparseDestroyCsrsv2Info()`**

8.1. cusparse<t>bsrmv()

```
cusparseStatus_t
cusparseSbsrmv(cusparseHandle_t         handle,
               cusparseDirection_t      dir,
               cusparseOperation_t      trans,
               int                      mb,
               int                      nb,
               int                      nnzb,
               const float*             alpha,
               const cusparseMatDescr_t descr,
               const float*             bsrVal,
               const int*               bsrRowPtr,
               const int*               bsrColInd,
               int                      blockDim,
               const float*             x,
               const float*             beta,
               float*                   y)

cusparseStatus_t
cusparseDbsrmv(cusparseHandle_t         handle,
               cusparseDirection_t      dir,
               cusparseOperation_t      trans,
               int                      mb,
               int                      nb,
               int                      nnzb,
               const double*            alpha,
               const cusparseMatDescr_t descr,
               const double*            bsrVal,
               const int*               bsrRowPtr,
               const int*               bsrColInd,
               int                      blockDim,
               const double*            x,
               const double*            beta,
               double*                  y)

cusparseStatus_t
cusparseCbsrmv(cusparseHandle_t         handle,
               cusparseDirection_t      dir,
               cusparseOperation_t      trans,
               int                      mb,
               int                      nb,
               int                      nnzb,
               const cuComplex*         alpha,
               const cusparseMatDescr_t descr,
               const cuComplex*         bsrVal,
               const int*               bsrRowPtr,
               const int*               bsrColInd,
               int                      blockDim,
               const cuComplex*         x,
               const cuComplex*         beta,
               cuComplex*               y)

cusparseStatus_t
cusparseZbsrmv(cusparseHandle_t         handle,
               cusparseDirection_t      dir,
               cusparseOperation_t      trans,
               int                      mb,
               int                      nb,
               int                      nnzb,
               const cuDoubleComplex*   alpha,
               const cusparseMatDescr_t descr,
               const cuDoubleComplex*   bsrVal,
               const int*               bsrRowPtr,
               const int*               bsrColInd,
```

This function performs the matrix-vector operation

$$y = \alpha * \text{op}(A) * x + \beta * y$$

where A is an $(mb * blockDim) \times (nb * blockDim)$ sparse matrix that is defined in BSR storage format by the three arrays **bsrVal**, **bsrRowPtr**, and **bsrColInd**); **x** and **y** are vectors; α and β are scalars; and

$$\text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

bsrmv() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Several comments on **bsrmv()**:

- Only **blockDim > 1** is supported
- Only **CUSPARSE_OPERATION_NON_TRANSPOSE** is supported, that is

$$y = \alpha * A * x + \beta * y$$

- Only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported.
- The size of vector **x** should be $(nb * blockDim)$ at least, and the size of vector **y** should be $(mb * blockDim)$ at least; otherwise, the kernel may return **CUSPARSE_STATUS_EXECUTION_FAILED** because of an out-of-bounds array.

For example, suppose the user has a CSR format and wants to try **bsrmv()**, the following code demonstrates how to use **csr2bsr()** conversion and **bsrmv()** multiplication in single precision.

```
// Suppose that A is m x n sparse matrix represented by CSR format,
// hx is a host vector of size n, and hy is also a host vector of size m.
// m and n are not multiple of blockDim.
// step 1: transform CSR to BSR with column-major order
int base, nnz;
int nnzb;
cusparseDirection_t dirA = CUSPARSE_DIRECTION_COLUMN;
int mb = (m + blockDim-1)/blockDim;
int nb = (n + blockDim-1)/blockDim;
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
cusparseXcsr2bsrNnz(handle, dirA, m, n,
        descrA, csrRowPtrA, csrColIndA, blockDim,
        descrC, bsrRowPtrC, &nnzb);
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(blockDim*blockDim)*nnzb);
cusparseScsr2bsr(handle, dirA, m, n,
        descrA, csrValA, csrRowPtrA, csrColIndA, blockDim,
        descrC, bsrValC, bsrRowPtrC, bsrColIndC);
// step 2: allocate vector x and vector y large enough for bsrmv
cudaMalloc((void**)&x, sizeof(float)*(nb*blockDim));
cudaMalloc((void**)&y, sizeof(float)*(mb*blockDim));
cudaMemcpy(x, hx, sizeof(float)*n, cudaMemcpyHostToDevice);
cudaMemcpy(y, hy, sizeof(float)*m, cudaMemcpyHostToDevice);
// step 3: perform bsrmv
cusparseSbsrmv(handle, dirA, transA, mb, nb, nnzb, &alpha,
   descrC, bsrValC, bsrRowPtrC, bsrColIndC, blockDim, x, &beta, y);
```

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
trans	the operation $\mathrm{op}(A)$. Only **CUSPARSE_OPERATION_NON_TRANSPOSE** is supported.
mb	number of block rows of matrix A.
nb	number of block columns of matrix A.
nnzb	number of nonzero blocks of matrix A.
alpha	<type> scalar used for multiplication.
descr	the descriptor of matrix A. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrVal	<type> array of **nnz** $($= **csrRowPtrA(mb)** − **csrRowPtrA(0)** $)$ nonzero blocks of matrix A.
bsrRowPtr	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColInd	integer array of **nnz** $($= **csrRowPtrA(mb)** − **csrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix A.
blockDim	block dimension of sparse matrix A, larger than zero.
x	<type> vector of $nb * blockDim$ elements.
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
y	<type> vector of $mb * blockDim$ elements.

Output

y	<type> updated vector.

See `cusparseStatus_t` for the description of the return status

8.2. cusparse<t>bsrxmv()

```
cusparseStatus_t
cusparseSbsrxmv(cusparseHandle_t         handle,
                cusparseDirection_t      dir,
                cusparseOperation_t      trans,
                int                      sizeOfMask,
                int                      mb,
                int                      nb,
                int                      nnzb,
                const float*             alpha,
                const cusparseMatDescr_t descr,
                const float*             bsrVal,
                const int*               bsrMaskPtr,
                const int*               bsrRowPtr,
                const int*               bsrEndPtr,
                const int*               bsrColInd,
                int                      blockDim,
                const float*             x,
                const float*             beta,
                float*                   y)

cusparseStatus_t
cusparseDbsrxmv(cusparseHandle_t         handle,
                cusparseDirection_t      dir,
                cusparseOperation_t      trans,
                int                      sizeOfMask,
                int                      mb,
                int                      nb,
                int                      nnzb,
                const double*            alpha,
                const cusparseMatDescr_t descr,
                const double*            bsrVal,
                const int*               bsrMaskPtr,
                const int*               bsrRowPtr,
                const int*               bsrEndPtr,
                const int*               bsrColInd,
                int                      blockDim,
                const double*            x,
                const double*            beta,
                double*                  y)

cusparseStatus_t
cusparseCbsrxmv(cusparseHandle_t         handle,
                cusparseDirection_t      dir,
                cusparseOperation_t      trans,
                int                      sizeOfMask,
                int                      mb,
                int                      nb,
                int                      nnzb,
                const cuComplex*         alpha,
                const cusparseMatDescr_t descr,
                const cuComplex*         bsrVal,
                const int*               bsrMaskPtr,
                const int*               bsrRowPtr,
                const int*               bsrEndPtr,
                const int*               bsrColInd,
                int                      blockDim,
                const cuComplex*         x,
                const cuComplex*         beta,
                cuComplex*               y)

cusparseStatus_t
cusparseZbsrxmv(cusparseHandle_t         handle,
                cusparseDirection_t      dir,
```

This function performs a **bsrmv** and a mask operation

$$y(\text{mask}) = (\alpha * \text{op}(A) * x + \beta * y)(\text{mask})$$

where A is an $(mb * blockDim) \times (nb * blockDim)$ sparse matrix that is defined in BSRX storage format by the four arrays **bsrVal**, **bsrRowPtr**, **bsrEndPtr**, and **bsrColInd**); **x** and **y** are vectors; α and β are scalars; and

$$\text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

The mask operation is defined by array **bsrMaskPtr** which contains updated block row indices of y. If row i is not specified in **bsrMaskPtr**, then **bsrxmv()** does not touch row block i of A and y.

For example, consider the 2×3 block matrix A:

$$A = \begin{bmatrix} A_{11} & A_{12} & O \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

and its one-based BSR format (three vector form) is

$$\begin{aligned} \text{bsrVal} &= [A_{11} \ \ A_{12} \ \ A_{21} \ \ A_{22} \ \ A_{23}] \\ \text{bsrRowPtr} &= [1 \ \ 3 \ \ 6] \\ \text{bsrColInd} &= [1 \ \ 2 \ \ 1 \ \ 2 \ \ 3] \end{aligned}$$

Suppose we want to do the following **bsrmv** operation on a matrix $\overline{A}$ which is slightly different from A.

$$\begin{bmatrix} y_1 \\ y_2 \end{bmatrix} := alpha * \left(\tilde{A} = \begin{bmatrix} O & O & O \\ O & A_{22} & O \end{bmatrix}\right) * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} y_1 \\ beta * y_2 \end{bmatrix}$$

We don't need to create another BSR format for the new matrix $\overline{A}$, all that we should do is to keep **bsrVal** and **bsrColInd** unchanged, but modify **bsrRowPtr** and add an additional array **bsrEndPtr** which points to the last nonzero elements per row of $\overline{A}$ plus 1.

For example, the following **bsrRowPtr** and **bsrEndPtr** can represent matrix $\overline{A}$:

$$\begin{aligned} \text{bsrRowPtr} &= [1 \ \ 4] \\ \text{bsrEndPtr} &= [1 \ \ 5] \end{aligned}$$

Further we can use a mask operator (specified by array **bsrMaskPtr**) to update particular block row indices of y only because y_1 is never changed. In this case, **bsrMaskPtr** $=$ [2] and **sizeOfMask**=1.

The mask operator is equivalent to the following operation:

$$\begin{bmatrix} ? \\ y_2 \end{bmatrix} := alpha * \begin{bmatrix} ? & ? & ? \\ O & A_{22} & O \end{bmatrix} * \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + beta * \begin{bmatrix} ? \\ y_2 \end{bmatrix}$$

If a block row is not present in the **bsrMaskPtr**, then no calculation is performed on that row, and the corresponding value in **y** is unmodified. The question mark "?" is used to inidcate row blocks not in **bsrMaskPtr**.

In this case, first row block is not present in **bsrMaskPtr**, so **bsrRowPtr[0]** and **bsrEndPtr[0]** are not touched also.

$$\begin{aligned} \text{bsrRowPtr} &= [? \quad 4] \\ \text{bsrEndPtr} &= [? \quad 5] \end{aligned}$$

bsrxmv() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

A couple of comments on **bsrxmv()**:

- Only **blockDim > 1** is supported
- Only **CUSPARSE_OPERATION_NON_TRANSPOSE** and **CUSPARSE_MATRIX_TYPE_GENERAL** are supported.
- Parameters **bsrMaskPtr**, **bsrRowPtr**, **bsrEndPtr** and **bsrColInd** are consistent with base index, either one-based or zero-based. The above example is one-based.

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
trans	the operation $\text{op}(A)$. Only **CUSPARSE_OPERATION_NON_TRANSPOSE** is supported.
sizeOfMask	number of updated block rows of y.
mb	number of block rows of matrix A.
nb	number of block columns of matrix A.
nnzb	number of nonzero blocks of matrix A.
alpha	<type> scalar used for multiplication.
descr	the descriptor of matrix A. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrVal	<type> array of **nnz** nonzero blocks of matrix A.
bsrMaskPtr	integer array of **sizeOfMask** elements that contains the indices corresponding to updated block rows.
bsrRowPtr	integer array of **mb** elements that contains the start of every block row.

bsrEndPtr	integer array of **mb** elements that contains the end of the every block row plus one.
bsrColInd	integer array of **nnzb** column indices of the nonzero blocks of matrix A.
blockDim	block dimension of sparse matrix A, larger than zero.
x	<type> vector of $nb * blockDim$ elements.
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
y	<type> vector of $mb * blockDim$ elements.

See `cusparseStatus_t` for the description of the return status

8.3. cusparse<t>bsrsv2_bufferSize()

```
cusparseStatus_t
cusparseSbsrsv2_bufferSize(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           cusparseOperation_t      transA,
                           int                      mb,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           float*                   bsrValA,
                           const int*               bsrRowPtrA,
                           const int*               bsrColIndA,
                           int                      blockDim,
                           bsrsv2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseDbsrsv2_bufferSize(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           cusparseOperation_t      transA,
                           int                      mb,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           double*                  bsrValA,
                           const int*               bsrRowPtrA,
                           const int*               bsrColIndA,
                           int                      blockDim,
                           bsrsv2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseCbsrsv2_bufferSize(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           cusparseOperation_t      transA,
                           int                      mb,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           cuComplex*               bsrValA,
                           const int*               bsrRowPtrA,
                           const int*               bsrColIndA,
                           int                      blockDim,
                           bsrsv2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseZbsrsv2_bufferSize(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           cusparseOperation_t      transA,
                           int                      mb,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex*         bsrValA,
                           const int*               bsrRowPtrA,
                           const int*               bsrColIndA,
                           int                      blockDim,
                           bsrsv2Info_t             info,
                           int*                     pBufferSizeInBytes)
```

This function returns size of the buffer used in **bsrsv2**, a new sparse triangular linear system **op(A)*y =** α **x**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Although there are six combinations in terms of parameter **trans** and the upper (lower) triangular part of **A**, **bsrsv2_bufferSize()** returns the maximum size buffer among these combinations. The buffer size depends on the dimensions **mb**, **blockDim**, and the number of nonzero blocks of the matrix **nnzb**. If the user changes the matrix, it is necessary to call **bsrsv2_bufferSize()** again to have the correct buffer size; otherwise a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation $\text{op}(A)$.
mb	number of block rows of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** $(=$ **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** $(=$ **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A; must be larger than zero.

Output

info	record of internal states based on different algorithms.

pBufferSizeInBytes	number of bytes of the buffer used in the **bsrsv2_analysis()** and **bsrsv2_solve()**.

See cusparseStatus_t for the description of the return status

8.4. cusparse<t>bsrsv2_analysis()

```
cusparseStatus_t
cusparseSbsrsv2_analysis(cusparseHandle_t         handle,
                         cusparseDirection_t      dirA,
                         cusparseOperation_t      transA,
                         int                      mb,
                         int                      nnzb,
                         const cusparseMatDescr_t descrA,
                         const float*             bsrValA,
                         const int*               bsrRowPtrA,
                         const int*               bsrColIndA,
                         int                      blockDim,
                         bsrsv2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseDbsrsv2_analysis(cusparseHandle_t         handle,
                         cusparseDirection_t      dirA,
                         cusparseOperation_t      transA,
                         int                      mb,
                         int                      nnzb,
                         const cusparseMatDescr_t descrA,
                         const double*            bsrValA,
                         const int*               bsrRowPtrA,
                         const int*               bsrColIndA,
                         int                      blockDim,
                         bsrsv2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseDbsrsv2_analysis(cusparseHandle_t         handle,
                         cusparseDirection_t      dirA,
                         cusparseOperation_t      transA,
                         int                      mb,
                         int                      nnzb,
                         const cusparseMatDescr_t descrA,
                         const cuComplex*         bsrValA,
                         const int*               bsrRowPtrA,
                         const int*               bsrColIndA,
                         int                      blockDim,
                         bsrsv2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseZbsrsv2_analysis(cusparseHandle_t         handle,
                         cusparseDirection_t      dirA,
                         cusparseOperation_t      transA,
                         int                      mb,
                         int                      nnzb,
                         const cusparseMatDescr_t descrA,
                         const cuDoubleComplex*   bsrValA,
                         const int*               bsrRowPtrA,
                         const int*               bsrColIndA,
                         int                      blockDim,
                         bsrsv2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)
```

This function performs the analysis phase of **bsrsv2**, a new sparse triangular linear system **op(A)*y =** α **x**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **x** and **y** are the right-hand side and the solution vectors; α is a scalar; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

The block of BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a buffer size returned by **bsrsv2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrsv2_analysis()** reports a structural zero and computes level information, which stored in the opaque structure **info**. The level information can extract more parallelism for a triangular solver. However **bsrsv2_solve()** can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **bsrsv2_analysis()** always reports the first structural zero, even when parameter **policy** is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. No structural zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if block **A(j,j)** is missing for some **j**. The user needs to call **cusparseXbsrsv2_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **bsrsv2_solve()** if **bsrsv2_analysis()** reports a structural zero. In this case, the user can still call **bsrsv2_solve()**, which will return a numerical zero at the same position as a structural zero. However the result **x** is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation $\mathrm{op}(A)$.
mb	number of block rows of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.

descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** $(=$ **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** $(=$ **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.
info	structure initialized using **cusparseCreateBsrsv2Info()**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is return by **bsrsv2_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

See `cusparseStatus_t` for the description of the return status

8.5. cusparse<t>bsrsv2_solve()

```
cusparseStatus_t
cusparseSbsrsv2_solve(cusparseHandle_t         handle,
                      cusparseDirection_t      dirA,
                      cusparseOperation_t      transA,
                      int                      mb,
                      int                      nnzb,
                      const float*             alpha,
                      const cusparseMatDescr_t descrA,
                      const float*             bsrValA,
                      const int*               bsrRowPtrA,
                      const int*               bsrColIndA,
                      int                      blockDim,
                      bsrsv2Info_t             info,
                      const float*             x,
                      float*                   y,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseDbsrsv2_solve(cusparseHandle_t         handle,
                      cusparseDirection_t      dirA,
                      cusparseOperation_t      transA,
                      int                      mb,
                      int                      nnzb,
                      const double*            alpha,
                      const cusparseMatDescr_t descrA,
                      const double*            bsrValA,
                      const int*               bsrRowPtrA,
                      const int*               bsrColIndA,
                      int                      blockDim,
                      bsrsv2Info_t             info,
                      const double*            x,
                      double*                  y,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseCbsrsv2_solve(cusparseHandle_t         handle,
                      cusparseDirection_t      dirA,
                      cusparseOperation_t      transA,
                      int                      mb,
                      int                      nnzb,
                      const cuComplex*         alpha,
                      const cusparseMatDescr_t descrA,
                      const cuComplex*         bsrValA,
                      const int*               bsrRowPtrA,
                      const int*               bsrColIndA,
                      int                      blockDim,
                      bsrsv2Info_t             info,
                      const cuComplex*         x,
                      cuComplex*               y,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseZbsrsv2_solve(cusparseHandle_t         handle,
                      cusparseDirection_t      dirA,
                      cusparseOperation_t      transA,
                      int                      mb,
                      int                      nnzb,
                      const cuDoubleComplex*   alpha,
                      const cusparseMatDescr_t descrA,
                      const cuDoubleComplex*   bsrValA,
```

This function performs the solve phase of **bsrsv2**, a new sparse triangular linear system **op(A)*y =** α **x**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored. Function **bsrsv02_solve()** can support an arbitrary **blockDim**.

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires a buffer size returned by **bsrsv2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **bsrsv2_solve()** can be done without level information, the user still needs to be aware of consistency. If **bsrsv2_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **bsrsv2_solve()** can be run with or without levels. On the other hand, if **bsrsv2_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **bsrsv2_solve()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The level information may not improve the performance, but may spend extra time doing analysis. For example, a tridiagonal matrix has no parallelism. In this case, **CUSPARSE_SOLVE_POLICY_NO_LEVEL** performs better than **CUSPARSE_SOLVE_POLICY_USE_LEVEL**. If the user has an iterative solver, the best approach is to do **bsrsv2_analysis()** with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** once. Then do **bsrsv2_solve()** with **CUSPARSE_SOLVE_POLICY_NO_LEVEL** in the first run, and with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** in the second run, and pick the fastest one to perform the remaining iterations.

Function **bsrsv02_solve()** has the same behavior as **csrsv02_solve()**. That is, **bsr2csr(bsrsv02(A)) = csrsv02(bsr2csr(A))**. The numerical zero of **csrsv02_solve()** means there exists some zero **A(j,j)**. The numerical zero of **bsrsv02_solve()** means there exists some block **A(j,j)** that is not invertible.

Function **bsrsv2_solve()** reports the first numerical zero, including a structural zero. No numerical zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is not invertible for some **j**. The user needs to call **cusparseXbsrsv2_zeroPivot()** to know where the numerical zero is.

The function supports the following properties if **pBuffer != NULL**

- The routine requires no extra storage

- The routine supports asynchronous execution
- The routine supports CUDA graph capture

For example, suppose L is a lower triangular matrix with unit diagonal, then the following code solves **L*y=x** by level information.

```
// Suppose that L is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// L is lower triangular with unit diagonal.
// Assumption:
// - dimension of matrix L is m(=mb*blockDim),
// - matrix L has nnz(=nnzb*blockDim*blockDim) nonzero elements,
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of L on device memory,
// - d_x is right hand side vector on device memory.
// - d_y is solution vector on device memory.
// - d_x and d_y are of size m.
cusparseMatDescr_t descr = 0;
bsrsv2Info_t info = 0;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

// step 1: create a descriptor which contains
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal, specified by parameter CUSPARSE_DIAG_TYPE_UNIT
//   (L may not have all diagonal elements.)
cusparseCreateMatDescr(&descr);
cusparseSetMatIndexBase(descr, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatFillMode(descr, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr, CUSPARSE_DIAG_TYPE_UNIT);

// step 2: create a empty info structure
cusparseCreateBsrsv2Info(&info);

// step 3: query how much memory used in bsrsv2, and allocate the buffer
cusparseDbsrsv2_bufferSize(handle, dir, trans, mb, nnzb, descr,
    d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, &pBufferSize);

// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis
cusparseDbsrsv2_analysis(handle, dir, trans, mb, nnzb, descr,
    d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim,
    info, policy, pBuffer);
// L has unit diagonal, so no structural zero is reported.
status = cusparseXbsrsv2_zeroPivot(handle, info, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is missing\n", structural_zero, structural_zero);
}

// step 5: solve L*y = x
cusparseDbsrsv2_solve(handle, dir, trans, mb, nnzb, &alpha, descr,
   d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info,
   d_x, d_y, policy, pBuffer);
// L has unit diagonal, so no numerical zero is reported.
status = cusparseXbsrsv2_zeroPivot(handle, info, &numerical_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is zero\n", numerical_zero, numerical_zero);
}

// step 6: free resources
cudaFree(pBuffer);
cusparseDestroyBsrsv2Info(info);
cusparseDestroyMatDescr(descr);
cusparseDestroy(handle);
```

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation $\mathrm{op}(A)$.
mb	number of block rows and block columns of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** $(=$ **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** $(=$ **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**, larger than zero.
info	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
x	<type> right-hand-side vector of size **m**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **bsrsv2_bufferSize()**.

Output

y	<type> solution vector of size **m**.

See `cusparseStatus_t` for the description of the return status

8.6. cusparseXbsrsv2_zeroPivot()

```
cusparseStatus_t
cusparseXbsrsv2_zeroPivot(cusparseHandle_t handle,
                          bsrsv2Info_t     info,
                          int*             position)
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** is either structural zero or numerical zero (singular block). Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXbsrsv2_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
info	**info** contains a structural zero or numerical zero if the user already called **bsrsv2_analysis()** or **bsrsv2_solve()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

See cusparseStatus_t for the description of the return status

8.7. cusparseCsrmvEx()

```
cusparseStatus_t
cusparseCsrmvEx_bufferSize(cusparseHandle_t         handle,
                           cusparseAlgMode_t        alg,
                           cusparseOperation_t      transA,
                           int                      m,
                           int                      n,
                           int                      nnz,
                           const void*              alpha,
                           cudaDataType             alphatype,
                           const cusparseMatDescr_t descrA,
                           const void*              csrValA,
                           cudaDataType             csrValAtype,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           const void*              x,
                           cudaDataType             xtype,
                           const void*              beta,
                           cudaDataType             betatype,
                           void*                    y,
                           cudaDataType             ytype,
                           cudaDataType             executiontype,
                           size_t*                  bufferSizeInBytes)

cusparseStatus_t
cusparseCsrmvEx(cusparseHandle_t         handle,
                cusparseAlgMode_t        alg,
                cusparseOperation_t      transA,
                int                      m,
                int                      n,
                int                      nnz,
                const void*              alpha,
                cudaDataType             alphatype,
                const cusparseMatDescr_t descrA,
                const void*              csrValA,
                cudaDataType             csrValAtype,
                const int*               csrRowPtrA,
                const int*               csrColIndA,
                const void*              x,
                cudaDataType             xtype,
                const void*              beta,
                cudaDataType             betatype,
                void*                    y,
                cudaDataType             ytype,
                cudaDataType             executiontype,
                void*                    buffer)
```

This function performs the matrix-vector operation

$$\mathrm{y} = \alpha * \mathrm{op}(A) * \mathrm{x} + \beta * \mathrm{y}$$

A is an **m×n** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are vectors;

The function **cusparseCsrmvEx_bufferSize** returns the size of the workspace needed by **cusparseCsrmvEx**.

The function has the following limitations:

- All pointers should be aligned with 128 bytes
- Only **CUSPARSE_OPERATION_NON_TRANSPOSE** operation is supported
- Only **CUSPARSE_MATRIX_TYPE_GENERAL** matrix type is supported
- Only **CUSPARSE_INDEX_BASE_ZERO** indexing is supported
- Half-precision is not supported
- The minimum GPU architecture supported is SM_53

The function has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input specifically required by cusparseCsrmvEx

alg	Algorithm implementation for csrmv, see **cusparseAlgMode_t** for possible values.
alphatype	Data type of **alpha**.
csrValAtype	Data type of **csrValA**.
xtype	Data type of **x**.
betatype	Data type of **beta**.
ytype	Data type of **y**.
executiontype	Data type used for computation.
bufferSizeInBytes	Pointer to a **size_t** variable, which will be assigned with the size of workspace needed by **cusparseCsrmvEx**.
buffer	Pointer to workspace buffer

See cusparseStatus_t for the description of the return status

8.8. cusparse<t>csrsv2_bufferSize()

```
cusparseStatus_t
cusparseScsrsv2_bufferSize(cusparseHandle_t         handle,
                           cusparseOperation_t      transA,
                           int                      m,
                           int                      nnz,
                           const cusparseMatDescr_t descrA,
                           float*                   csrValA,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           csrsv2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsrsv2_bufferSize(cusparseHandle_t         handle,
                           cusparseOperation_t      transA,
                           int                      m,
                           int                      nnz,
                           const cusparseMatDescr_t descrA,
                           double*                  csrValA,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           csrsv2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseCcsrsv2_bufferSize(cusparseHandle_t         handle,
                           cusparseOperation_t      transA,
                           int                      m,
                           int                      nnz,
                           const cusparseMatDescr_t descrA,
                           cuComplex*               csrValA,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           csrsv2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseZcsrsv2_bufferSize(cusparseHandle_t         handle,
                           cusparseOperation_t      transA,
                           int                      m,
                           int                      nnz,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex*         csrValA,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           csrsv2Info_t             info,
                           int*                     pBufferSizeInBytes)
```

This function returns the size of the buffer used in **csrsv2**, a new sparse triangular linear system **op(A)*y** $= \alpha$ **x**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Although there are six combinations in terms of the parameter **trans** and the upper (lower) triangular part of **A**, **csrsv2_bufferSize()** returns the maximum size buffer of these combinations. The buffer size depends on the dimension and the number of nonzero elements of the matrix. If the user changes the matrix, it is necessary to call **csrsv2_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\text{op}(A)$.
m	number of rows of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** ($=$ **csrRowPtrA(m)** $-$ **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** $+1$ elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** ($=$ **csrRowPtrA(m)** $-$ **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

Output

info	record of internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in the **csrsv2_analysis** and **csrsv2_solve**.

See `cusparseStatus_t` for the description of the return status

8.9. cusparse<t>csrsv2_analysis()

```
cusparseStatus_t
cusparseScsrsv2_analysis(cusparseHandle_t         handle,
                         cusparseOperation_t      transA,
                         int                      m,
                         int                      nnz,
                         const cusparseMatDescr_t descrA,
                         const float*             csrValA,
                         const int*               csrRowPtrA,
                         const int*               csrColIndA,
                         csrsv2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseDcsrsv2_analysis(cusparseHandle_t         handle,
                         cusparseOperation_t      transA,
                         int                      m,
                         int                      nnz,
                         const cusparseMatDescr_t descrA,
                         const double*            csrValA,
                         const int*               csrRowPtrA,
                         const int*               csrColIndA,
                         csrsv2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseCcsrsv2_analysis(cusparseHandle_t         handle,
                         cusparseOperation_t      transA,
                         int                      m,
                         int                      nnz,
                         const cusparseMatDescr_t descrA,
                         const cuComplex*         csrValA,
                         const int*               csrRowPtrA,
                         const int*               csrColIndA,
                         csrsv2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseZcsrsv2_analysis(cusparseHandle_t         handle,
                         cusparseOperation_t      transA,
                         int                      m,
                         int                      nnz,
                         const cusparseMatDescr_t descrA,
                         const cuDoubleComplex*   csrValA,
                         const int*               csrRowPtrA,
                         const int*               csrColIndA,
                         csrsv2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)
```

This function performs the analysis phase of **csrsv2**, a new sparse triangular linear system **op(A)*y =** α **x**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a buffer size returned by **csrsv2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csrsv2_analysis()** reports a structural zero and computes level information that is stored in opaque structure **info**. The level information can extract more parallelism for a triangular solver. However **csrsv2_solve()** can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **csrsv2_analysis()** always reports the first structural zero, even if the policy is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. No structural zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is missing for some **j**. The user needs to call **cusparseXcsrsv2_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **csrsv2_solve()** if **csrsv2_analysis()** reports a structural zero. In this case, the user can still call **csrsv2_solve()** which will return a numerical zero in the same position as the structural zero. However the result **x** is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
transA	the operation $\mathrm{op}(A)$.
m	number of rows of matrix **A**.
nnz	number of nonzero elements of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** $+1$ elements that contains the start of every row and the end of the last row plus one.

csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
info	structure initialized using **cusparseCreateCsrsv2Info()**.
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **csrsv2_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

See `cusparseStatus_t` for the description of the return status

8.10. cusparse<t>csrsv2_solve()

```
cusparseStatus_t
cusparseScsrsv2_solve(cusparseHandle_t         handle,
                      cusparseOperation_t      transA,
                      int                      m,
                      int                      nnz,
                      const float*             alpha,
                      const cusparseMatDescr_t descra,
                      const float*             csrValA,
                      const int*               csrRowPtrA,
                      const int*               csrColIndA,
                      csrsv2Info_t             info,
                      const float*             x,
                      float*                   y,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseDcsrsv2_solve(cusparseHandle_t         handle,
                      cusparseOperation_t      transA,
                      int                      m,
                      int                      nnz,
                      const double*            alpha,
                      const cusparseMatDescr_t descra,
                      const double*            csrValA,
                      const int*               csrRowPtrA,
                      const int*               csrColIndA,
                      csrsv2Info_t             info,
                      const double*            x,
                      double*                  y,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseCcsrsv2_solve(cusparseHandle_t         handle,
                      cusparseOperation_t      transA,
                      int                      m,
                      int                      nnz,
                      const cuComplex*         alpha,
                      const cusparseMatDescr_t descra,
                      const cuComplex*         csrValA,
                      const int*               csrRowPtrA,
                      const int*               csrColIndA,
                      csrsv2Info_t             info,
                      const cuComplex*         x,
                      cuComplex*               y,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseZcsrsv2_solve(cusparseHandle_t         handle,
                      cusparseOperation_t      transA,
                      int                      m,
                      int                      nnz,
                      const cuDoubleComplex*   alpha,
                      const cusparseMatDescr_t descra,
                      const cuDoubleComplex*   csrValA,
                      const int*               csrRowPtrA,
                      const int*               csrColIndA,
                      csrsv2Info_t             info,
                      const cuDoubleComplex*   x,
                      cuDoubleComplex*         y,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)
```

This function performs the solve phase of **csrsv2**, a new sparse triangular linear system **op(A)*y =** α **x**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **x** and **y** are the right-hand-side and the solution vectors; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires the buffer size returned by **csrsv2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **csrsv2_solve()** can be done without level information, the user still needs to be aware of consistency. If **csrsv2_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **csrsv2_solve()** can be run with or without levels. On the contrary, if **csrsv2_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **csrsv2_solve()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The level information may not improve the performance but spend extra time doing analysis. For example, a tridiagonal matrix has no parallelism. In this case, **CUSPARSE_SOLVE_POLICY_NO_LEVEL** performs better than **CUSPARSE_SOLVE_POLICY_USE_LEVEL**. If the user has an iterative solver, the best approach is to do **csrsv2_analysis()** with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** once. Then do **csrsv2_solve()** with **CUSPARSE_SOLVE_POLICY_NO_LEVEL** in the first run and with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** in the second run, picking faster one to perform the remaining iterations.

Function **csrsv2_solve()** reports the first numerical zero, including a structural zero. If **status** is 0, no numerical zero was found. Furthermore, no numerical zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is zero for some **j**. The user needs to call **cusparseXcsrsv2_zeroPivot()** to know where the numerical zero is.

For example, suppose L is a lower triangular matrix with unit diagonal, the following code solves **L*y=x** by level information.

```
// Suppose that L is m x m sparse matrix represented by CSR format,
// L is lower triangular with unit diagonal.
// Assumption:
// - dimension of matrix L is m,
// - matrix L has nnz number zero elements,
// - handle is already created by cusparseCreate(),
// - (d_csrRowPtr, d_csrColInd, d_csrVal) is CSR of L on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.

cusparseMatDescr_t descr = 0;
csrsv2Info_t info = 0;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans = CUSPARSE_OPERATION_NON_TRANSPOSE;

// step 1: create a descriptor which contains
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal, specified by parameter CUSPARSE_DIAG_TYPE_UNIT
//   (L may not have all diagonal elements.)
cusparseCreateMatDescr(&descr);
cusparseSetMatIndexBase(descr, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatFillMode(descr, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr, CUSPARSE_DIAG_TYPE_UNIT);

// step 2: create a empty info structure
cusparseCreateCsrsv2Info(&info);

// step 3: query how much memory used in csrsv2, and allocate the buffer
cusparseDcsrsv2_bufferSize(handle, trans, m, nnz, descr,
    d_csrVal, d_csrRowPtr, d_csrColInd, &pBufferSize);
// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis
cusparseDcsrsv2_analysis(handle, trans, m, nnz, descr,
    d_csrVal, d_csrRowPtr, d_csrColInd,
    info, policy, pBuffer);
// L has unit diagonal, so no structural zero is reported.
status = cusparseXcsrsv2_zeroPivot(handle, info, &structural_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is missing\n", structural_zero, structural_zero);
}

// step 5: solve L*y = x
cusparseDcsrsv2_solve(handle, trans, m, nnz, &alpha, descr,
   d_csrVal, d_csrRowPtr, d_csrColInd, info,
   d_x, d_y, policy, pBuffer);
// L has unit diagonal, so no numerical zero is reported.
status = cusparseXcsrsv2_zeroPivot(handle, info, &numerical_zero);
if (CUSPARSE_STATUS_ZERO_PIVOT == status){
   printf("L(%d,%d) is zero\n", numerical_zero, numerical_zero);
}

// step 6: free resources
cudaFree(pBuffer);
cusparseDestroyCsrsv2Info(info);
cusparseDestroyMatDescr(descr);
cusparseDestroy(handle);
```

Remark: **`csrsv2_solve()`** needs more nonzeros per row to achieve good performance. It would perform better if more than 16 nonzeros per row in average.

The function supports the following properties if **`pBuffer != NULL`**

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`transA`	the operation $\text{op}(A)$.
`m`	number of rows and columns of matrix **`A`**.
`alpha`	<type> scalar used for multiplication.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**, while the supported diagonal types are **`CUSPARSE_DIAG_TYPE_UNIT`** and **`CUSPARSE_DIAG_TYPE_NON_UNIT`**.
`csrValA`	<type> array of **`nnz`** $($= **`csrRowPtrA(m)`** $-$ **`csrRowPtrA(0)`** $)$ nonzero elements of matrix **`A`**.
`csrRowPtrA`	integer array of **`m`** $+1$ elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of **`nnz`** $($= **`csrRowPtrA(m)`** $-$ **`csrRowPtrA(0)`** $)$ column indices of the nonzero elements of matrix **`A`**.
`info`	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
`x`	<type> right-hand-side vector of size **`m`**.
`policy`	The supported policies are **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`** and **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**.
`pBuffer`	buffer allocated by the user, the size is return by **`csrsv2_bufferSize`**.

Output

`y`	<type> solution vector of size **`m`**.

See `cusparseStatus_t` for the description of the return status

8.11. cusparseXcsrsv2_zeroPivot()

```
cusparseStatus_t
cusparseXcsrsv2_zeroPivot(cusparseHandle_t handle,
                          csrsv2Info_t     info,
                          int*             position)
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero. Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXcsrsv2_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **csrsv2_analysis()** or **csrsv2_solve()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise, if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

See cusparseStatus_t for the description of the return status

8.12. cusparse<t>gemvi()

```
cusparseStatus_t
cusparseSgemvi_bufferSize(cusparseHandle_t    handle,
                          cusparseOperation_t transA,
                          int                 m,
                          int                 n,
                          int                 nnz,
                          int*                pBufferSize)

cusparseStatus_t
cusparseDgemvi_bufferSize(cusparseHandle_t    handle,
                          cusparseOperation_t transA,
                          int                 m,
                          int                 n,
                          int                 nnz,
                          int*                pBufferSize)

cusparseStatus_t
cusparseCgemvi_bufferSize(cusparseHandle_t    handle,
                          cusparseOperation_t transA,
                          int                 m,
                          int                 n,
                          int                 nnz,
                          int*                pBufferSize)

cusparseStatus_t
cusparseZgemvi_bufferSize(cusparseHandle_t    handle,
                          cusparseOperation_t transA,
                          int                 m,
                          int                 n,
                          int                 nnz,
                          int*                pBufferSize)
```

```
cusparseStatus_t
cusparseSgemvi(cusparseHandle_t     handle,
               cusparseOperation_t  transA,
               int                  m,
               int                  n,
               const float*         alpha,
               const float*         A,
               int                  lda,
               int                  nnz,
               const float*         x,
               const int*           xInd,
               const float*         beta,
               float*               y,
               cusparseIndexBase_t  idxBase,
               void*                pBuffer)

cusparseStatus_t
cusparseDgemvi(cusparseHandle_t     handle,
               cusparseOperation_t  transA,
               int                  m,
               int                  n,
               const double*        alpha,
               const double*        A,
               int                  lda,
               int                  nnz,
               const double*        x,
               const int*           xInd,
               const float*         beta,
               double*              y,
               cusparseIndexBase_t  idxBase,
```

This function performs the matrix-vector operation

$$y = \alpha * \mathrm{op}(A) * x + \beta * y$$

`A` is an **`m×n`** dense matrix and a sparse vector x that is defined in a sparse storage format by the two arrays **`xVal`**, **`xInd`** of length **`nnz`**, and **`y`** is a dense vector; α and β are scalars; and

$$\mathrm{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

To simplify the implementation, we have not (yet) optimized the transpose multiple case. We recommend the following for users interested in this case.

1. Convert the matrix from CSR to CSC format using one of the **`csr2csc()`** functions. Notice that by interchanging the rows and columns of the result you are implicitly transposing the matrix.

2. Call the **`gemvi()`** function with the **`cusparseOperation_t`** parameter set to **`CUSPARSE_OPERATION_NON_TRANSPOSE`** and with the interchanged rows and columns of the matrix stored in CSC format. This (implicitly) multiplies the vector by the transpose of the matrix in the original CSR format.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

The function **`cusparse<t>gemvi_bufferSize()`** returns size of buffer used in **`cusparse<t>gemvi()`**

Input

`handle`	handle to the cuSPARSE library context.
`trans`	the operation $\mathrm{op}(A)$.
`m`	number of rows of matrix **`A`**.
`n`	number of columns of matrix **`A`**.
`alpha`	<type> scalar used for multiplication.
`A`	the pointer to dense matrix **`A`**.
`lda`	size of the leading dimension of **`A`**.
`nnz`	number of nonzero elements of vector **`x`**.
`x`	<type> sparse vector of **`nnz`** elements of size **`n`** if $\mathrm{op}(A) = A$, and size **`m`** if $\mathrm{op}(A) = A^T$ or $\mathrm{op}(A) = A^H$
`xInd`	Indices of non-zero values in **`x`**
`beta`	<type> scalar used for multiplication. If **`beta`** is zero, **`y`** does not have to be a valid input.
`y`	<type> dense vector of **`m`** elements if $\mathrm{op}(A) = A$, and **`n`** elements if $\mathrm{op}(A) = A^T$ or $\mathrm{op}(A) = A^H$

idxBase	0 or 1, for 0 based or 1 based indexing, respectively
pBufferSize	number of elements needed the buffer used in **cusparse<t>gemvi().**
pBuffer	working space buffer

Output

y	<type> updated dense vector.

See `cusparseStatus_t` for the description of the return status

Chapter 9. CUSPARSE LEVEL 3 FUNCTION REFERENCE

This chapter describes sparse linear algebra functions that perform operations between sparse and (usually tall) dense matrices.

In particular, the solution of sparse triangular linear systems with multiple right-hand sides is implemented in two phases. First, during the analysis phase, the sparse triangular matrix is analyzed to determine the dependencies between its elements by calling the appropriate **`csrsm2_analysis()`** function. The analysis is specific to the sparsity pattern of the given matrix and to the selected **`cusparseOperation_t`** type. The information from the analysis phase is stored in the parameter of type **`csrsm2Info_t`** that has been initialized previously with a call to **`cusparseCreateCsrsm2Info()`**.

Second, during the solve phase, the given sparse triangular linear system is solved using the information stored in the **`csrsm2Info_t`** parameter by calling the appropriate **`csrsm2_solve()`** function. The solve phase may be performed multiple times with different multiple right-hand sides, while the analysis phase needs to be performed only once. This is especially useful when a sparse triangular linear system must be solved for different sets of multiple right-hand sides one at a time, while its coefficient matrix remains the same.

Finally, once all the solves have completed, the opaque data structure pointed to by the **`csrsm2Info_t`** parameter can be released by calling **`cusparseDestroyCsrsm2Info()`**.

9.1. cusparse<t>bsrmm()

```
cusparseStatus_t
cusparseSbsrmm(cusparseHandle_t         handle,
               cusparseDirection_t      dirA,
               cusparseOperation_t      transA,
               cusparseOperation_t      transB,
               int                      mb,
               int                      n,
               int                      kb,
               int                      nnzb,
               const float*             alpha,
               const cusparseMatDescr_t descrA,
               const float*             bsrValA,
               const int*               bsrRowPtrA,
               const int*               bsrColIndA,
               int                      blockDim,
               const float*             B,
               int                      ldb,
               const float*             beta,
               float*                   C,
               int                      ldc)

cusparseStatus_t
cusparseDbsrmm(cusparseHandle_t         handle,
               cusparseDirection_t      dirA,
               cusparseOperation_t      transA,
               cusparseOperation_t      transB,
               int                      mb,
               int                      n,
               int                      kb,
               int                      nnzb,
               const double*            alpha,
               const cusparseMatDescr_t descrA,
               const double*            bsrValA,
               const int*               bsrRowPtrA,
               const int*               bsrColIndA,
               int                      blockDim,
               const double*            B,
               int                      ldb,
               const double*            beta,
               double*                  C,
               int                      ldc)

cusparseStatus_t
cusparseCbsrmm(cusparseHandle_t         handle,
               cusparseDirection_t      dirA,
               cusparseOperation_t      transA,
               cusparseOperation_t      transB,
               int                      mb,
               int                      n,
               int                      kb,
               int                      nnzb,
               const cuComplex*         alpha,
               const cusparseMatDescr_t descrA,
               const cuComplex*         bsrValA,
               const int*               bsrRowPtrA,
               const int*               bsrColIndA,
               int                      blockDim,
               const cuComplex*         B,
               int                      ldb,
               const cuComplex*         beta,
               cuComplex*               C,
               int                      ldc)
```

This function performs one of the following matrix-matrix operations:

$$C = \alpha * \text{op}(A) * \text{op}(B) + \beta * C$$

A is an **mb×kb** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**; **B** and **C** are dense matrices; α and β are scalars; and

$$\text{op}(A) = \begin{cases} A & \text{if transA} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if transA} == \text{CUSPARSE\_OPERATION\_TRANSPOSE (not supported)} \\ A^H & \text{if transA} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE (not supported)} \end{cases}$$

and

$$\text{op}(B) = \begin{cases} B & \text{if transB} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T & \text{if transB} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ B^H & \text{if transB} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE (not supported)} \end{cases}$$

The function has the following limitations:

- Only **CUSPARSE_MATRIX_TYPE_GENERAL** matrix type is supported
- Only **blockDim > 1** is supported

The motivation of **transpose(B)** is to improve memory access of matrix **B**. The computational pattern of **A*transpose(B)** with matrix **B** in column-major order is equivalent to **A*B** with matrix **B** in row-major order.

In practice, no operation in an iterative solver or eigenvalue solver uses **A*transpose(B)**. However, we can perform **A*transpose(transpose(B))** which is the same as **A*B**. For example, suppose **A** is **mb*kb**, **B** is **k*n** and **C** is **m*n**, the following code shows usage of **cusparseDbsrmm()**.

```
// A is mb*kb, B is k*n and C is m*n
    const int m = mb*blockSize;
    const int k = kb*blockSize;
    const int ldb_B = k; // leading dimension of B
    const int ldc   = m; // leading dimension of C
// perform C:=alpha*A*B + beta*C
    cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL );
    cusparseDbsrmm(cusparse_handle,
               CUSPARSE_DIRECTION_COLUMN,
               CUSPARSE_OPERATION_NON_TRANSPOSE,
               CUSPARSE_OPERATION_NON_TRANSPOSE,
               mb, n, kb, nnzb, alpha,
               descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockSize,
               B, ldb_B,
               beta, C, ldc);
```

Instead of using **A*B**, our proposal is to transpose **B** to **Bt** by first calling **cublas<t>geam()**, and then to perform **A*transpose(Bt)**.

```
// step 1: Bt := transpose(B)
    const int m = mb*blockSize;
    const int k = kb*blockSize;
    double *Bt;
    const int ldb_Bt = n; // leading dimension of Bt
    cudaMalloc((void**)&Bt, sizeof(double)*ldb_Bt*k);
    double one  = 1.0;
    double zero = 0.0;
    cublasSetPointerMode(cublas_handle, CUBLAS_POINTER_MODE_HOST);
    cublasDgeam(cublas_handle, CUBLAS_OP_T, CUBLAS_OP_T,
        n, k, &one, B, int ldb_B, &zero, B, int ldb_B, Bt, ldb_Bt);

// step 2: perform C:=alpha*A*transpose(Bt) + beta*C
    cusparseDbsrmm(cusparse_handle,
                   CUSPARSE_DIRECTION_COLUMN,
                   CUSPARSE_OPERATION_NON_TRANSPOSE,
                   CUSPARSE_OPERATION_TRANSPOSE,
                   mb, n, kb, nnzb, alpha,
                   descrA, bsrValA, bsrRowPtrA, bsrColIndA, blockSize,
                   Bt, ldb_Bt,
                   beta, C, ldc);
```

bsrmm() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation **op(A)**.
transB	the operation **op(B)**.
mb	number of block rows of sparse matrix **A**.
n	number of columns of dense matrix **op(B)** and **A**.
kb	number of block columns of sparse matrix **A**.
nnzb	number of non-zero blocks of sparse matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** − **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.

bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** $($= **bsrRowPtrA(mb)** − **bsrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**, larger than zero.
B	array of dimensions **(ldb, n)** if **op(B)=B** and **(ldb, k)** otherwise.
ldb	leading dimension of **B**. If **op(B)=B**, it must be at least $\max(1, k)$ If **op(B) != B**, it must be at least **max(1, n)**.
beta	<type> scalar used for multiplication. If **beta** is zero, **C** does not have to be a valid input.
C	array of dimensions **(ldc, n)**.
ldc	leading dimension of **C**. It must be at least $\max(1, m)$ if **op(A)=A** and at least $\max(1, k)$ otherwise.

Output

C	<type> updated array of dimensions **(ldc, n)**.

See `cusparseStatus_t` for the description of the return status

9.2. cusparse<t>bsrsm2_bufferSize()

```
cusparseStatus_t
cusparseSbsrsm2_bufferSize(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           cusparseOperation_t      transA,
                           cusparseOperation_t      transX,
                           int                      mb,
                           int                      n,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           float*                   bsrSortedValA,
                           const int*               bsrSortedRowPtrA,
                           const int*               bsrSortedColIndA,
                           int                      blockDim,
                           bsrsm2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseDbsrsm2_bufferSize(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           cusparseOperation_t      transA,
                           cusparseOperation_t      transX,
                           int                      mb,
                           int                      n,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           double*                  bsrSortedValA,
                           const int*               bsrSortedRowPtrA,
                           const int*               bsrSortedColIndA,
                           int                      blockDim,
                           bsrsm2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseCbsrsm2_bufferSize(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           cusparseOperation_t      transA,
                           cusparseOperation_t      transX,
                           int                      mb,
                           int                      n,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           cuComplex*               bsrSortedValA,
                           const int*               bsrSortedRowPtrA,
                           const int*               bsrSortedColIndA,
                           int                      blockDim,
                           bsrsm2Info_t             info,
                           int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseZbsrsm2_bufferSize(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           cusparseOperation_t      transA,
                           cusparseOperation_t      transX,
                           int                      mb,
                           int                      n,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex*         bsrSortedValA,
                           const int*               bsrSortedRowPtrA,
                           const int*               bsrSortedColIndA,
                           int                      blockDim,
                           bsrsm2Info_t             info,
                           int*                     pBufferSizeInBytes)
```

This function returns size of buffer used in **bsrsm2()**, a new sparse triangular linear system **op(A)*op(X)=** α **op(B)**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **B** and **X** are the right-hand-side and the solution matrices; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Although there are six combinations in terms of parameter **trans** and the upper (and lower) triangular part of **A**, **bsrsm2_bufferSize()** returns the maximum size of the buffer among these combinations. The buffer size depends on dimension **mb,blockDim** and the number of nonzeros of the matrix, **nnzb**. If the user changes the matrix, it is necessary to call **bsrsm2_bufferSize()** again to get the correct buffer size, otherwise a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
transA	the operation **op(A)**.
transX	the operation **op(X)**.
mb	number of block rows of matrix **A**.
n	number of columns of matrix **op(B)** and **op(X)**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**; larger than zero.

Output

`info`	record internal states based on different algorithms.
`pBufferSizeInBytes`	number of bytes of the buffer used in **`bsrsm2_analysis()`** and **`bsrsm2_solve()`**.

See `cusparseStatus_t` for the description of the return status

9.3. cusparse<t>bsrsm2_analysis()

```
cusparseStatus_t
cusparseSbsrsm2_analysis(cusparseHandle_t         handle,
                         cusparseDirection_t      dirA,
                         cusparseOperation_t      transA,
                         cusparseOperation_t      transX,
                         int                      mb,
                         int                      n,
                         int                      nnzb,
                         const cusparseMatDescr_t descrA,
                         const float*             bsrSortedVal,
                         const int*               bsrSortedRowPtr,
                         const int*               bsrSortedColInd,
                         int                      blockDim,
                         bsrsm2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseDbsrsm2_analysis(cusparseHandle_t         handle,
                         cusparseDirection_t      dirA,
                         cusparseOperation_t      transA,
                         cusparseOperation_t      transX,
                         int                      mb,
                         int                      n,
                         int                      nnzb,
                         const cusparseMatDescr_t descrA,
                         const double*            bsrSortedVal,
                         const int*               bsrSortedRowPtr,
                         const int*               bsrSortedColInd,
                         int                      blockDim,
                         bsrsm2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseCbsrsm2_analysis(cusparseHandle_t         handle,
                         cusparseDirection_t      dirA,
                         cusparseOperation_t      transA,
                         cusparseOperation_t      transX,
                         int                      mb,
                         int                      n,
                         int                      nnzb,
                         const cusparseMatDescr_t descrA,
                         const cuComplex*         bsrSortedVal,
                         const int*               bsrSortedRowPtr,
                         const int*               bsrSortedColInd,
                         int                      blockDim,
                         bsrsm2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseZbsrsm2_analysis(cusparseHandle_t         handle,
                         cusparseDirection_t      dirA,
                         cusparseOperation_t      transA,
                         cusparseOperation_t      transX,
                         int                      mb,
                         int                      n,
                         int                      nnzb,
                         const cusparseMatDescr_t descrA,
                         const cuDoubleComplex*   bsrSortedVal,
                         const int*               bsrSortedRowPtr,
                         const int*               bsrSortedColInd,
```

This function performs the analysis phase of **bsrsm2()**, a new sparse triangular linear system **op(A)*op(X) =** α **op(B)**.

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **B** and **X** are the right-hand-side and the solution matrices; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

and

$$\text{op}(X) = \begin{cases} X & \text{if transX == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ X^T & \text{if transX == CUSPARSE\_OPERATION\_TRANSPOSE} \\ X^H & \text{if transX == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE (not supported)} \end{cases}$$

and **op(B)** and **op(X)** are equal.

The block of BSR format is of size **blockDim*blockDim**, stored in column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires the buffer size returned by **bsrsm2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrsm2_analysis()** reports a structural zero and computes the level information stored in opaque structure **info**. The level information can extract more parallelism during a triangular solver. However **bsrsm2_solve()** can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **bsrsm2_analysis()** always reports the first structural zero, even if the parameter **policy** is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. Besides, no structural zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if block **A(j,j)** is missing for some **j**. The user must call **cusparseXbsrsm2_query_zero_pivot()** to know where the structural zero is.

If **bsrsm2_analysis()** reports a structural zero, the solve will return a numerical zero in the same position as the structural zero but this result **X** is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.

transA	the operation **op(A)**.
transX	the operation **op(B)** and **op(X)**.
mb	number of block rows of matrix **A**.
n	number of columns of matrix **op(B)** and **op(X)**.
nnzb	number of non-zero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.
bsrValA	<type> array of **nnzb** $($= **bsrRowPtrA(mb)** − **bsrRowPtrA(0)** $)$ nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** $($= **bsrRowPtrA(mb)** − **bsrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix **A**; larger than zero.
info	structure initialized using **cusparseCreateBsrsm2Info**.
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is return by **bsrsm2_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

See `cusparseStatus_t` for the description of the return status

9.4. cusparse<t>bsrsm2_solve()

```
cusparseStatus_t
cusparseSbsrsm2_solve(cusparseHandle_t         handle,
                      cusparseDirection_t      dirA,
                      cusparseOperation_t      transA,
                      cusparseOperation_t      transX,
                      int                      mb,
                      int                      n,
                      int                      nnzb,
                      const float*             alpha,
                      const cusparseMatDescr_t descrA,
                      const float*             bsrSortedVal,
                      const int*               bsrSortedRowPtr,
                      const int*               bsrSortedColInd,
                      int                      blockDim,
                      bsrsm2Info_t             info,
                      const float*             B,
                      int                      ldb,
                      float*                   X,
                      int                      ldx,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseDbsrsm2_solve(cusparseHandle_t         handle,
                      cusparseDirection_t      dirA,
                      cusparseOperation_t      transA,
                      cusparseOperation_t      transX,
                      int                      mb,
                      int                      n,
                      int                      nnzb,
                      const double*            alpha,
                      const cusparseMatDescr_t descrA,
                      const double*            bsrSortedVal,
                      const int*               bsrSortedRowPtr,
                      const int*               bsrSortedColInd,
                      int                      blockDim,
                      bsrsm2Info_t             info,
                      const double*            B,
                      int                      ldb,
                      double*                  X,
                      int                      ldx,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseCbsrsm2_solve(cusparseHandle_t         handle,
                      cusparseDirection_t      dirA,
                      cusparseOperation_t      transA,
                      cusparseOperation_t      transX,
                      int                      mb,
                      int                      n,
                      int                      nnzb,
                      const cuComplex*         alpha,
                      const cusparseMatDescr_t descrA,
                      const cuComplex*         bsrSortedVal,
                      const int*               bsrSortedRowPtr,
                      const int*               bsrSortedColInd,
                      int                      blockDim,
                      bsrsm2Info_t             info,
                      const cuComplex*         B,
                      int                      ldb,
                      cuComplex*               X,
                      int                      ldx,
```

This function performs the solve phase of the solution of a sparse triangular linear system:

$$\mathrm{op}(A) * \mathrm{op}(X) = \alpha * \mathrm{op}(B)$$

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**); **B** and **X** are the right-hand-side and the solution matrices; α is a scalar, and

$$\mathrm{op}(A) = \begin{cases} A & \text{if transA == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if transA == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if transA == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

and

$$\mathrm{op}(X) = \begin{cases} X & \text{if transX == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ X^T & \text{if transX == CUSPARSE\_OPERATION\_TRANSPOSE} \\ X^H & \text{not supported} \end{cases}$$

Only **op(A)=A** is supported.

op(B) and **op(X)** must be performed in the same way. In other words, if **op(B)=B**, **op(X)=X**.

The block of BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored. Function **bsrsm02_solve()** can support an arbitrary **blockDim**.

This function may be executed multiple times for a given matrix and a particular operation type.

This function requires the buffer size returned by **bsrsm2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **bsrsm2_solve()** can be done without level information, the user still needs to be aware of consistency. If **bsrsm2_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **bsrsm2_solve()** can be run with or without levels. On the other hand, if **bsrsm2_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **bsrsm2_solve()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrsm02_solve()** has the same behavior as **bsrsv02_solve()**, reporting the first numerical zero, including a structural zero. The user must call **cusparseXbsrsm2_query_zero_pivot()** to know where the numerical zero is.

The motivation of **transpose(X)** is to improve the memory access of matrix **X**. The computational pattern of **transpose(X)** with matrix **X** in column-major order is equivalent to **X** with matrix **X** in row-major order.

In-place is supported and requires that **B** and **X** point to the same memory block, and **ldb=ldx**.

The function supports the following properties if **pBuffer != NULL**

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`dirA`	storage format of blocks, either `CUSPARSE_DIRECTION_ROW` or `CUSPARSE_DIRECTION_COLUMN`.
`transA`	the operation `op(A)`.
`transX`	the operation `op(B)` and `op(X)`.
`mb`	number of block rows of matrix `A`.
`n`	number of columns of matrix `op(B)` and `op(X)`.
`nnzb`	number of non-zero blocks of matrix `A`.
`alpha`	<type> scalar used for multiplication.
`descrA`	the descriptor of matrix `A`. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL`, while the supported diagonal types are `CUSPARSE_DIAG_TYPE_UNIT` and `CUSPARSE_DIAG_TYPE_NON_UNIT`.
`bsrValA`	<type> array of `nnzb` $($= `bsrRowPtrA(mb)` $-$ `bsrRowPtrA(0)` $)$ non-zero blocks of matrix `A`.
`bsrRowPtrA`	integer array of `mb` $+1$ elements that contains the start of every block row and the end of the last block row plus one.
`bsrColIndA`	integer array of `nnzb` $($= `bsrRowPtrA(mb)` $-$ `bsrRowPtrA(0)` $)$ column indices of the nonzero blocks of matrix `A`.
`blockDim`	block dimension of sparse matrix `A`; larger than zero.
`info`	structure initialized using `cusparseCreateBsrsm2Info()`.
`B`	<type> right-hand-side array.
`ldb`	leading dimension of `B`. If `op(B)=B`, `ldb >= (mb*blockDim)`; otherwise, `ldb >= n`.
`ldx`	leading dimension of `X`. If `op(X)=X`, then `ldx >= (mb*blockDim)`. otherwise `ldx >= n`.
`policy`	the supported policies are `CUSPARSE_SOLVE_POLICY_NO_LEVEL` and `CUSPARSE_SOLVE_POLICY_USE_LEVEL`.
`pBuffer`	buffer allocated by the user; the size is returned by `bsrsm2_bufferSize()`.

Output

`X`	<type> solution array with leading dimensions `ldx`.

See `cusparseStatus_t` for the description of the return status

9.5. cusparseXbsrsm2_zeroPivot()

```
cusparseStatus_t
cusparseXbsrsm2_zeroPivot(cusparseHandle_t handle,
                          bsrsm2Info_t     info,
                          int*             position)
```

If the returned error code is **`CUSPARSE_STATUS_ZERO_PIVOT`**, **`position=j`** means **`A(j,j)`** is either a structural zero or a numerical zero (singular block). Otherwise **`position=-1`**.

The **`position`** can be 0-base or 1-base, the same as the matrix.

Function **`cusparseXbsrsm2_zeroPivot()`** is a blocking call. It calls **`cudaDeviceSynchronize()`** to make sure all previous kernels are done.

The **`position`** can be in the host memory or device memory. The user can set the proper mode with **`cusparseSetPointerMode()`**.

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`info`	**`info`** contains a structural zero or a numerical zero if the user already called **`bsrsm2_analysis()`** or **`bsrsm2_solve()`**.

Output

`position`	if no structural or numerical zero, **`position`** is -1; otherwise, if **`A(j,j)`** is missing or **`U(j,j)`** is zero, **`position=j`**.

See `cusparseStatus_t` for the description of the return status

9.6. cusparse<t>csrsm2_bufferSizeExt()

```
cusparseStatus_t
cusparseScsrsm2_bufferSizeExt(cusparseHandle_t         handle,
                              int                      algo,
                              cusparseOperation_t      transA,
                              cusparseOperation_t      transB,
                              int                      m,
                              int                      nrhs,
                              int                      nnz,
                              const float*             alpha,
                              const cusparseMatDescr_t descrA,
                              const float*             csrSortedValA,
                              const int*               csrSortedRowPtrA,
                              const int*               csrSortedColIndA,
                              const float*             B,
                              int                      ldb,
                              csrsm2Info_t             info,
                              cusparseSolvePolicy_t    policy,
                              size_t*                  pBufferSize)

cusparseStatus_t
cusparseDcsrsm2_bufferSizeExt(cusparseHandle_t         handle,
                              int                      algo,
                              cusparseOperation_t      transA,
                              cusparseOperation_t      transB,
                              int                      m,
                              int                      nrhs,
                              int                      nnz,
                              const double*            alpha,
                              const cusparseMatDescr_t descrA,
                              const double*            csrSortedValA,
                              const int*               csrSortedRowPtrA,
                              const int*               csrSortedColIndA,
                              const double*            B,
                              int                      ldb,
                              csrsm2Info_t             info,
                              cusparseSolvePolicy_t    policy,
                              size_t*                  pBufferSize)

cusparseStatus_t
cusparseCcsrsm2_bufferSizeExt(cusparseHandle_t         handle,
                              int                      algo,
                              cusparseOperation_t      transA,
                              cusparseOperation_t      transB,
                              int                      m,
                              int                      nrhs,
                              int                      nnz,
                              const cuComplex*         alpha,
                              const cusparseMatDescr_t descrA,
                              const cuComplex*         csrSortedValA,
                              const int*               csrSortedRowPtrA,
                              const int*               csrSortedColIndA,
                              const cuComplex*         B,
                              int                      ldb,
                              csrsm2Info_t             info,
                              cusparseSolvePolicy_t    policy,
                              size_t*                  pBufferSize)

cusparseStatus_t
cusparseZcsrsm2_bufferSizeExt(cusparseHandle_t         handle,
                              int                      algo,
                              cusparseOperation_t      transA,
                              cusparseOperation_t      transB,
                              int                      m,
```

This function returns the size of the buffer used in **`csrsm2`**, a sparse triangular linear system **`op(A) * op(X) =`** α **`op(B)`**.

`A` is an **`m×m`** sparse matrix that is defined in CSR storage format by the three arrays **`csrValA`**, **`csrRowPtrA`**, and **`csrColIndA`**); **`B`** and **`X`** are the right-hand-side matrix and the solution matrix; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`algo`	algo = 0 is non-block version; algo = 1 is block version.
`transA`	the operation $\text{op}(A)$.
`transB`	the operation $\text{op}(B)$.
`m`	number of rows of matrix **`A`**.
`nrhs`	number of columns of right hand side matrix **`op(B)`**.
`nnz`	number of nonzero elements of matrix **`A`**.
`alpha`	<type> scalar used for multiplication.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**, while the supported diagonal types are **`CUSPARSE_DIAG_TYPE_UNIT`** and **`CUSPARSE_DIAG_TYPE_NON_UNIT`**.
`csrValA`	<type> array of **`nnz`** (= **`csrRowPtrA(m)`** - **`csrRowPtrA(0)`**) nonzero elements of matrix **`A`**.
`csrRowPtrA`	integer array of **`m`** $+1$ elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of **`nnz`** (= **`csrRowPtrA(m)`** - **`csrRowPtrA(0)`**) column indices of the nonzero elements of matrix **`A`**.
`B`	<type> right-hand-side matrix. **`op(B)`** is of size **`m-by-nrhs`**.
`ldb`	leading dimension of **`B`** and **`X`**.
`info`	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
`policy`	The supported policies are **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`** and **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**.

Output

info	record of internal states based on different algorithms.
pBufferSize	number of bytes of the buffer used in the **csrsm2_analysis** and **csrsm2_solve**.

See `cusparseStatus_t` for the description of the return status

9.7. cusparse<t>csrsm2_analysis()

```
cusparseStatus_t
cusparseScsrsm2_analysis(cusparseHandle_t         handle,
                         int                      algo,
                         cusparseOperation_t      transA,
                         cusparseOperation_t      transB,
                         int                      m,
                         int                      nrhs,
                         int                      nnz,
                         const float*             alpha,
                         const cusparseMatDescr_t descrA,
                         const float*             csrSortedValA,
                         const int*               csrSortedRowPtrA,
                         const int*               csrSortedColIndA,
                         const float*             B,
                         int                      ldb,
                         csrsm2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseDcsrsm2_analysis(cusparseHandle_t         handle,
                         int                      algo,
                         cusparseOperation_t      transA,
                         cusparseOperation_t      transB,
                         int                      m,
                         int                      nrhs,
                         int                      nnz,
                         const double*            alpha,
                         const cusparseMatDescr_t descrA,
                         const double*            csrSortedValA,
                         const int*               csrSortedRowPtrA,
                         const int*               csrSortedColIndA,
                         const double*            B,
                         int                      ldb,
                         csrsm2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseCcsrsm2_analysis(cusparseHandle_t         handle,
                         int                      algo,
                         cusparseOperation_t      transA,
                         cusparseOperation_t      transB,
                         int                      m,
                         int                      nrhs,
                         int                      nnz,
                         const cuComplex*         alpha,
                         const cusparseMatDescr_t descrA,
                         const cuComplex*         csrSortedValA,
                         const int*               csrSortedRowPtrA,
                         const int*               csrSortedColIndA,
                         const cuComplex*         B,
                         int                      ldb,
                         csrsm2Info_t             info,
                         cusparseSolvePolicy_t    policy,
                         void*                    pBuffer)

cusparseStatus_t
cusparseZcsrsm2_analysis(cusparseHandle_t         handle,
                         int                      algo,
                         cusparseOperation_t      transA,
                         cusparseOperation_t      transB,
                         int                      m,
```

This function performs the analysis phase of **csrsm2**, a sparse triangular linear system **op(A) * op(X) =** α **op(B)**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **B** and **X** are the right-hand-side matrix and the solution matrix; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if trans} == \text{CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if trans} == \text{CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if trans} == \text{CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

It is expected that this function will be executed only once for a given matrix and a particular operation type.

This function requires a buffer size returned by **csrsm2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csrsm2_analysis()** reports a structural zero and computes level information that is stored in opaque structure **info**. The level information can extract more parallelism for a triangular solver. However **csrsm2_solve()** can be done without level information. To disable level information, the user needs to specify the policy of the triangular solver as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **csrsm2_analysis()** always reports the first structural zero, even if the policy is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. No structural zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is missing for some **j**. The user needs to call **cusparseXcsrsm2_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **csrsm2_solve()** if **csrsm2_analysis()** reports a structural zero. In this case, the user can still call **csrsm2_solve()** which will return a numerical zero in the same position as the structural zero. However the result **X** is meaningless.

Input

handle	handle to the cuSPARSE library context.
algo	algo = 0 is non-block version; algo = 1 is block version.
transA	the operation $\text{op}(A)$.
transB	the operation $\text{op}(B)$.
m	number of rows of matrix **A**.
nrhs	number of columns of matrix **op(B)**.
nnz	number of nonzero elements of matrix **A**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, while the supported diagonal types are **CUSPARSE_DIAG_TYPE_UNIT** and **CUSPARSE_DIAG_TYPE_NON_UNIT**.

csrValA	<type> array of **nnz** $(=$ **csrRowPtrA(m) - csrRowPtrA(0)** $)$ nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** $+1$ elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** $(=$ **csrRowPtrA(m) - csrRowPtrA(0)** $)$ column indices of the nonzero elements of matrix **A**.
B	<type> right-hand-side matrix. **op(B)** is of size **m-by-nrhs**.
ldb	leading dimension of **B** and **X**.
info	structure initialized using **cusparseCreateCsrsv2Info()**.
policy	The supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **csrsm2_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

See `cusparseStatus_t` for the description of the return status

9.8. cusparse<t>csrsm2_solve()

```
cusparseStatus_t
cusparseScsrsm2_solve(cusparseHandle_t         handle,
                      int                      algo,
                      cusparseOperation_t      transA,
                      cusparseOperation_t      transB,
                      int                      m,
                      int                      nrhs,
                      int                      nnz,
                      const float*             alpha,
                      const cusparseMatDescr_t descrA,
                      const float*             csrSortedValA,
                      const int*               csrSortedRowPtrA,
                      const int*               csrSortedColIndA,
                      float*                   B,
                      int                      ldb,
                      csrsm2Info_t             info,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseDcsrsm2_solve(cusparseHandle_t         handle,
                      int                      algo,
                      cusparseOperation_t      transA,
                      cusparseOperation_t      transB,
                      int                      m,
                      int                      nrhs,
                      int                      nnz,
                      const double*            alpha,
                      const cusparseMatDescr_t descrA,
                      const double*            csrSortedValA,
                      const int*               csrSortedRowPtrA,
                      const int*               csrSortedColIndA,
                      double*                  B,
                      int                      ldb,
                      csrsm2Info_t             info,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseCcsrsm2_solve(cusparseHandle_t         handle,
                      int                      algo,
                      cusparseOperation_t      transA,
                      cusparseOperation_t      transB,
                      int                      m,
                      int                      nrhs,
                      int                      nnz,
                      const cuComplex*         alpha,
                      const cusparseMatDescr_t descrA,
                      const cuComplex*         csrSortedValA,
                      const int*               csrSortedRowPtrA,
                      const int*               csrSortedColIndA,
                      cuComplex*               B,
                      int                      ldb,
                      csrsm2Info_t             info,
                      cusparseSolvePolicy_t    policy,
                      void*                    pBuffer)

cusparseStatus_t
cusparseZcsrsm2_solve(cusparseHandle_t         handle,
                      int                      algo,
                      cusparseOperation_t      transA,
                      cusparseOperation_t      transB,
                      int                      m,
```

This function performs the solve phase of **csrsm2**, a sparse triangular linear system **op(A) * op(X) =** α **op(B)**.

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**); **B** and **X** are the right-hand-side matrix and the solution matrix; α is a scalar; and

$$\text{op}(A) = \begin{cases} A & \text{if transA == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if transA == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if transA == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

transB acts on both matrix **B** and matrix **X**, only **CUSPARSE_OPERATION_NON_TRANSPOSE** and **CUSPARSE_OPERATION_TRANSPOSE**. The operation is in-place, matrix **B** is overwritten by matrix **X**.

ldb must be not less than **m** if **transB** = **CUSPARSE_OPERATION_NON_TRANSPOSE**. Otherwise, **ldb** must be not less than **nrhs**.

This function requires the buffer size returned by **csrsm2_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **csrsm2_solve()** can be done without level information, the user still needs to be aware of consistency. If **csrsm2_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **csrsm2_solve()** can be run with or without levels. On the contrary, if **csrsm2_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **csrsm2_solve()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The level information may not improve the performance but spend extra time doing analysis. For example, a tridiagonal matrix has no parallelism. In this case, **CUSPARSE_SOLVE_POLICY_NO_LEVEL** performs better than **CUSPARSE_SOLVE_POLICY_USE_LEVEL**. If the user has an iterative solver, the best approach is to do **csrsm2_analysis()** with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** once. Then do **csrsm2_solve()** with **CUSPARSE_SOLVE_POLICY_NO_LEVEL** in the first run and with **CUSPARSE_SOLVE_POLICY_USE_LEVEL** in the second run, picking faster one to perform the remaining iterations.

Function **csrsm2_solve()** reports the first numerical zero, including a structural zero. If **status** is 0, no numerical zero was found. Furthermore, no numerical zero is reported if **CUSPARSE_DIAG_TYPE_UNIT** is specified, even if **A(j,j)** is zero for some **j**. The user needs to call **cusparseXcsrsm2_zeroPivot()** to know where the numerical zero is.

csrsm2 provides two algorithms specified by the parameter **algo**. **algo=0** is non-block version and **algo=1** is block version. non-block version is memory-bound, limited by bandwidth. block version partitions the matrix into small tiles and applies desne operations. Although it has more flops than non-block version, it may be faster if non-block version already reaches maximum bandwidth..

Appendix section shows an example of csrsm2.

The function supports the following properties if **pBuffer != NULL**

- The routine requires no extra storage

- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`algo`	algo = 0 is non-block version; algo = 1 is block version.
`transA`	the operation $\mathrm{op}(A)$.
`transB`	the operation $\mathrm{op}(B)$.
`m`	number of rows and columns of matrix **A**.
`nrhs`	number of columns of matrix `op(B)`.
`nnz`	number of nonzeros of matrix **A**.
`alpha`	<type> scalar used for multiplication.
`descrA`	the descriptor of matrix **A**. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL`, while the supported diagonal types are `CUSPARSE_DIAG_TYPE_UNIT` and `CUSPARSE_DIAG_TYPE_NON_UNIT`.
`csrValA`	<type> array of `nnz` $(=$ `csrRowPtrA(m)` $-$ `csrRowPtrA(0)` $)$ nonzero elements of matrix **A**.
`csrRowPtrA`	integer array of `m` $+1$ elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of `nnz` $(=$ `csrRowPtrA(m)` $-$ `csrRowPtrA(0)` $)$ column indices of the nonzero elements of matrix **A**.
`B`	<type> right-hand-side matrix. `op(B)` is of size `m-by-nrhs`.
`ldb`	leading dimension of `B` and `X`.
`info`	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
`policy`	The supported policies are `CUSPARSE_SOLVE_POLICY_NO_LEVEL` and `CUSPARSE_SOLVE_POLICY_USE_LEVEL`.
`pBuffer`	buffer allocated by the user, the size is returned by `csrsm2_bufferSize`.

Output

`X`	<type> solution matrix, `op(X)` is of size `m-by-nrhs`.

See `cusparseStatus_t` for the description of the return status

9.9. cusparseXcsrsm2_zeroPivot()

```
cusparseStatus_t
cusparseXcsrsm2_zeroPivot(cusparseHandle_t handle,
                          csrsm2Info_t     info,
                          int*             position)
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero. Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXcsrsm2_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **csrsm2_analysis()** or **csrsm2_solve()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise, if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

See cusparseStatus_t for the description of the return status

9.10. cusparse<t>gemmi() [DEPRECATED]

[[DEPRECATED]] use `cusparseSpMM()` instead. *The routine will be removed in the next major release*

```
cusparseStatus_t
cusparseSgemmi(cusparseHandle_t handle,
               int              m,
               int              n,
               int              k,
               int              nnz,
               const float*     alpha,
               const float*     A,
               int              lda,
               const float*     cscValB,
               const int*       cscColPtrB,
               const int*       cscRowIndB,
               const float*     beta,
               float*           C,
               int              ldc)

cusparseStatus_t
cusparseDgemmi(cusparseHandle_t handle,
               int              m,
               int              n,
               int              k,
               int              nnz,
               const double*    alpha,
               const double*    A,
               int              lda,
               const double*    cscValB,
               const int*       cscColPtrB,
               const int*       cscRowIndB,
               const double*    beta,
               double*          C,
               int              ldc)

cusparseStatus_t
cusparseCgemmi(cusparseHandle_t handle,
               int              m,
               int              n,
               int              k,
               int              nnz,
               const cuComplex* alpha,
               const cuComplex* A,
               int              lda,
               const cuComplex* cscValB,
               const int*       cscColPtrB,
               const int*       cscRowIndB,
               const cuComplex* beta,
               cuComplex*       C,
               int              ldc)

cusparseStatus_t
cusparseZgemmi(cusparseHandle_t       handle,
               int                    m,
               int                    n,
               int                    k,
               int                    nnz,
               const cuDoubleComplex* alpha,
               const cuDoubleComplex* A,
               int                    lda,
               const cuDoubleComplex* cscValB,
               const int*             cscColPtrB,
               const int*             cscRowIndB,
```

This function performs the following matrix-matrix operations:

$$C = \alpha * A * B + \beta * C$$

A and **C** are dense matrices; **B** is a **k×n** sparse matrix that is defined in CSC storage format by the three arrays **cscValB**, **cscColPtrB**, and **cscRowIndB**); α and β are scalars; and

Remark: **B** is base-0.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
n	number of columns of matrices **B** and **C**.
k	number of columns of matrix **A**.
nnz	number of nonzero elements of sparse matrix **B**.
alpha	<type> scalar used for multiplication.
A	array of dimensions **(lda, k)**.
lda	leading dimension of **A**. It must be at least **m**
cscValB	<type> array of **nnz** (= **cscColPtrB(k)** - **cscColPtrB(0)**) nonzero elements of matrix **B**.
cscColPtrB	integer array of **k** + 1 elements that contains the start of every row and the end of the last row plus one.
cscRowIndB	integer array of **nnz** (= **cscColPtrB(k)** - **cscColPtrB(0)**) column indices of the nonzero elements of matrix **B**.
beta	<type> scalar used for multiplication. If **beta** is zero, **C** does not have to be a valid input.
C	array of dimensions **(ldc, n)**.
ldc	leading dimension of **C**. It must be at least **m**

Output

C	<type> updated array of dimensions **(ldc, n)**.

See `cusparseStatus_t` for the description of the return status

Chapter 10. CUSPARSE EXTRA FUNCTION REFERENCE

This chapter describes the extra routines used to manipulate sparse matrices.

10.1. cusparse<t>csrgeam2()

```
cusparseStatus_t
cusparseScsrgeam2_bufferSizeExt(cusparseHandle_t         handle,
                                int                      m,
                                int                      n,
                                const float*             alpha,
                                const cusparseMatDescr_t descrA,
                                int                      nnzA,
                                const float*             csrSortedValA,
                                const int*
 csrSortedRowPtrA,
                                const int*
 csrSortedColIndA,
                                const float*             beta,
                                const cusparseMatDescr_t descrB,
                                int                      nnzB,
                                const float*             csrSortedValB,
                                const int*
 csrSortedRowPtrB,
                                const int*
 csrSortedColIndB,
                                const cusparseMatDescr_t descrC,
                                const float*             csrSortedValC,
                                const int*
 csrSortedRowPtrC,
                                const int*
 csrSortedColIndC,
                                size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsrgeam2_bufferSizeExt(cusparseHandle_t         handle,
                                int                      m,
                                int                      n,
                                const double*            alpha,
                                const cusparseMatDescr_t descrA,
                                int                      nnzA,
                                const double*            csrSortedValA,
                                const int*
 csrSortedRowPtrA,
                                const int*
 csrSortedColIndA,
                                const double*            beta,
                                const cusparseMatDescr_t descrB,
                                int                      nnzB,
                                const double*            csrSortedValB,
                                const int*
 csrSortedRowPtrB,
                                const int*
 csrSortedColIndB,
                                const cusparseMatDescr_t descrC,
                                const double*            csrSortedValC,
                                const int*
 csrSortedRowPtrC,
                                const int*
 csrSortedColIndC,
                                size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseCcsrgeam2_bufferSizeExt(cusparseHandle_t         handle,
                                int                      m,
                                int                      n,
                                const cuComplex*         alpha,
```

This function performs following matrix-matrix operation

$$C = \alpha * A + \beta * B$$

where **A**, **B**, and **C** are **m×n** sparse matrices (defined in CSR storage format by the three arrays **csrValA|csrValB|csrValC**, **csrRowPtrA|csrRowPtrB|csrRowPtrC**, and **csrColIndA|csrColIndB|csrcolIndC** respectively), and α and β are scalars. Since **A** and **B** have different sparsity patterns, cuSPARSE adopts a two-step approach to complete sparse matrix **C**. In the first step, the user allocates **csrRowPtrC** of **m +1**elements and uses function **cusparseXcsrgeam2Nnz()** to determine **csrRowPtrC** and the total number of nonzero elements. In the second step, the user gathers **nnzC** (number of nonzero elements of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC(m)-csrRowPtrC(0))** and allocates **csrValC, csrColIndC** of **nnzC** elements respectively, then finally calls function **cusparse[S|D|C|Z]csrgeam2()** to complete matrix **C**.

The general procedure is as follows:

```
int baseC, nnzC;
/* alpha, nnzTotalDevHostPtr points to host memory */
size_t BufferSizeInBytes;
char *buffer = NULL;
int *nnzTotalDevHostPtr = &nnzC;
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
/* prepare buffer */
cusparseScsrgeam2_bufferSizeExt(handle, m, n,
    alpha,
    descrA, nnzA,
    csrValA, csrRowPtrA, csrColIndA,
    beta,
    descrB, nnzB,
    csrValB, csrRowPtrB, csrColIndB,
    descrC,
    csrValC, csrRowPtrC, csrColIndC
    &bufferSizeInBytes
    );
cudaMalloc((void**)&buffer, sizeof(char)*bufferSizeInBytes);
cusparseXcsrgeam2Nnz(handle, m, n,
        descrA, nnzA, csrRowPtrA, csrColIndA,
        descrB, nnzB, csrRowPtrB, csrColIndB,
        descrC, csrRowPtrC, nnzTotalDevHostPtr,
        buffer);
if (NULL != nnzTotalDevHostPtr){
    nnzC = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzC, csrRowPtrC+m, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&baseC, csrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzC -= baseC;
}
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnzC);
cudaMalloc((void**)&csrValC, sizeof(float)*nnzC);
cusparseScsrgeam2(handle, m, n,
        alpha,
        descrA, nnzA,
        csrValA, csrRowPtrA, csrColIndA,
        beta,
        descrB, nnzB,
        csrValB, csrRowPtrB, csrColIndB,
        descrC,
        csrValC, csrRowPtrC, csrColIndC
        buffer);
```

Several comments on **csrgeam2()**:

- The other three combinations, NT, TN, and TT, are not supported by cuSPARSE. In order to do any one of the three, the user should use the routine **csr2csc()** to convert $A|B$ to $A^T|B^T$.
- Only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported. If either **A** or **B** is symmetric or Hermitian, then the user must extend the matrix to a full one and reconfigure the **MatrixType** field of the descriptor to **CUSPARSE_MATRIX_TYPE_GENERAL**.
- If the sparsity pattern of matrix **C** is known, the user can skip the call to function **cusparseXcsrgeam2Nnz()**. For example, suppose that the user has an iterative algorithm which would update **A** and **B** iteratively but keep the sparsity patterns. The user can call function **cusparseXcsrgeam2Nnz()** once to set up the sparsity pattern of **C**, then call function **cusparse[S|D|C|Z]geam()** only for each iteration.
- The pointers **alpha** and **beta** must be valid.
- When **alpha** or **beta** is zero, it is not considered a special case by cuSPARSE. The sparsity pattern of **C** is independent of the value of **alpha** and **beta**. If the user wants $C = 0 \times A + 1 \times B^T$, then **csr2csc()** is better than **csrgeam2()**.
- **csrgeam2()** is the same as **csrgeam()** except **csrgeam2()** needs explicit buffer where **csrgeam()** allocates the buffer internally.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	number of rows of sparse matrix **A,B,C**.
n	number of columns of sparse matrix **A,B,C**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzA	number of nonzero elements of sparse matrix **A**.
csrValA	<type> array of **nnzA** ($=$ **csrRowPtrA(m)** $-$ **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** $+1$ elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnzA** ($=$ **csrRowPtrA(m)** $-$ **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.
beta	<type> scalar used for multiplication. If **beta** is zero, **y** does not have to be a valid input.
descrB	the descriptor of matrix **B**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzB	number of nonzero elements of sparse matrix **B**.

csrValB	<type> array of **nnzB** $(=$ **csrRowPtrB(m)** $-$ **csrRowPtrB(0)** $)$ nonzero elements of matrix **B**.
csrRowPtrB	integer array of **m** $+1$ elements that contains the start of every row and the end of the last row plus one.
csrColIndB	integer array of **nnzB** $(=$ **csrRowPtrB(m)** $-$ **csrRowPtrB(0)** $)$ column indices of the nonzero elements of matrix **B**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.

Output

csrValC	<type> array of **nnzC** $(=$ **csrRowPtrC(m)** $-$ **csrRowPtrC(0)** $)$ nonzero elements of matrix **C**.
csrRowPtrC	integer array of **m** $+1$ elements that contains the start of every row and the end of the last row plus one.
csrColIndC	integer array of **nnzC** $(=$ **csrRowPtrC(m)** $-$ **csrRowPtrC(0)** $)$ column indices of the nonzero elements of matrix**C**.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory. It is equal to **(csrRowPtrC(m)-csrRowPtrC(0))**.

See `cusparseStatus_t` for the description of the return status

10.2. cusparse<t>csrgemm2() [DEPRECATED]

[[DEPRECATED]] use **cusparseSpGEMM()** instead. *The routine will be removed in the next major release*

```
cusparseStatus_t
cusparseScsrgemm2_bufferSizeExt(cusparseHandle_t         handle,
                                int                      m,
                                int                      n,
                                int                      k,
                                const float*             alpha,
                                const cusparseMatDescr_t descrA,
                                int                      nnzA,
                                const int*               csrRowPtrA,
                                const int*               csrColIndA,
                                const cusparseMatDescr_t descrB,
                                int                      nnzB,
                                const int*               csrRowPtrB,
                                const int*               csrColIndB,
                                const float*             beta,
                                const cusparseMatDescr_t descrD,
                                int                      nnzD,
                                const int*               csrRowPtrD,
                                const int*               csrColIndD,
                                csrgemm2Info_t           info,
                                size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsrgemm2_bufferSizeExt(cusparseHandle_t         handle,
                                int                      m,
                                int                      n,
                                int                      k,
                                const double*            alpha,
                                const cusparseMatDescr_t descrA,
                                int                      nnzA,
                                const int*               csrRowPtrA,
                                const int*               csrColIndA,
                                const cusparseMatDescr_t descrB,
                                int                      nnzB,
                                const int*               csrRowPtrB,
                                const int*               csrColIndB,
                                const double*            beta,
                                const cusparseMatDescr_t descrD,
                                int                      nnzD,
                                const int*               csrRowPtrD,
                                const int*               csrColIndD,
                                csrgemm2Info_t           info,
                                size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseCcsrgemm2_bufferSizeExt(cusparseHandle_t         handle,
                                int                      m,
                                int                      n,
                                int                      k,
                                const cuComplex*         alpha,
                                const cusparseMatDescr_t descrA,
                                int                      nnzA,
                                const int*               csrRowPtrA,
                                const int*               csrColIndA,
                                const cusparseMatDescr_t descrB,
                                int                      nnzB,
                                const int*               csrRowPtrB,
                                const int*               csrColIndB,
```

This function performs following matrix-matrix operation:

$$C = \text{alpha} * A * B + \text{beta} * D$$

where **A**, **B**, **D** and **C** are **m×k**, **k×n**, **m×n** and **m×n** sparse matrices (defined in CSR storage format by the three arrays **csrValA|csrValB|csrValD|csrValC**, **csrRowPtrA|csrRowPtrB|csrRowPtrD|csrRowPtrC**, and **csrColIndA|csrColIndB|csrColIndD|csrcolIndC** respectively.

The **csrgemm2** uses **alpha** and **beta** to support the following operations:

alpha	beta	operation
NULL	NULL	invalid
NULL	!NULL	C = beta*D, A and B are not used
!NULL	NULL	C = alpha*A*B, D is not used
!NULL	!NULL	C = alpha*A*B + beta*D

The numerical value of **alpha** and **beta** only affects the numerical values of **C**, not its sparsity pattern. For example, if **alpha** and **beta** are not zero, the sparsity pattern of **C** is union of **A*B** and **D**, independent of numerical value of **alpha** and **beta**.

The following table shows different operations according to the value of **m**, **n** and **k**

m,n,k	operation
m<0 or n <0 or k<0	invalid
m is 0 or n is 0	do nothing
m >0 and n >0 and k is 0	invalid if beta is zero; C = beta*D if beta is not zero.
m >0 and n >0 and k >0	C = beta*D if alpha is zero. C = alpha*A*B if beta is zero. C = alpha*A*B + beta*D if alpha and beta are not zero.

This function requires the buffer size returned by **csrgemm2_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The cuSPARSE library adopts a two-step approach to complete sparse matrix. In the first step, the user allocates **csrRowPtrC** of **m+1** elements and uses the function **cusparseXcsrgemm2Nnz()** to determine **csrRowPtrC** and the total number of nonzero elements. In the second step, the user gathers **nnzC** (the number of nonzero elements of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC(m)-csrRowPtrC(0))** and allocates **csrValC** and **csrColIndC** of **nnzC** elements respectively, then finally calls function **cusparse[S|D|C|Z]csrgemm2()** to evaluate matrix **C**.

The general procedure of **C=-A*B+D** is as follows:

```
// assume matrices A, B and D are ready.
int baseC, nnzC;
csrgemm2Info_t info = NULL;
size_t bufferSize;
void *buffer = NULL;
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzC;
double alpha = -1.0;
double beta  =  1.0;
cusparseSetPointerMode(handle, CUSPARSE_POINTER_MODE_HOST);

// step 1: create an opaque structure
cusparseCreateCsrgemm2Info(&info);

// step 2: allocate buffer for csrgemm2Nnz and csrgemm2
cusparseDcsrgemm2_bufferSizeExt(handle, m, n, k, &alpha,
    descrA, nnzA, csrRowPtrA, csrColIndA,
    descrB, nnzB, csrRowPtrB, csrColIndB,
    &beta,
    descrD, nnzD, csrRowPtrD, csrColIndD,
    info,
    &bufferSize);
cudaMalloc(&buffer, bufferSize);

// step 3: compute csrRowPtrC
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cusparseXcsrgemm2Nnz(handle, m, n, k,
        descrA, nnzA, csrRowPtrA, csrColIndA,
        descrB, nnzB, csrRowPtrB, csrColIndB,
        descrD, nnzD, csrRowPtrD, csrColIndD,
        descrC, csrRowPtrC, nnzTotalDevHostPtr,
        info, buffer );
if (NULL != nnzTotalDevHostPtr){
    nnzC = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzC, csrRowPtrC+m, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&baseC, csrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzC -= baseC;
}

// step 4: finish sparsity pattern and value of C
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnzC);
cudaMalloc((void**)&csrValC, sizeof(double)*nnzC);
// Remark: set csrValC to null if only sparsity pattern is required.
cusparseDcsrgemm2(handle, m, n, k, &alpha,
        descrA, nnzA, csrValA, csrRowPtrA, csrColIndA,
        descrB, nnzB, csrValB, csrRowPtrB, csrColIndB,
        &beta,
        descrD, nnzD, csrValD, csrRowPtrD, csrColIndD,
        descrC, csrValC, csrRowPtrC, csrColIndC,
        info, buffer);

// step 5: destroy the opaque structure
cusparseDestroyCsrgemm2Info(info);
```

Several comments on **csrgemm2()**:

- Only the NN version is supported. For other modes, the user has to transpose **A** or **B** explicitly.
- Only **CUSPARSE_MATRIX_TYPE_GENERAL** is supported. If either **A** or **B** is symmetric or Hermitian, the user must extend the matrix to a full one and reconfigure the **MatrixType** field descriptor to **CUSPARSE_MATRIX_TYPE_GENERAL**.

- if **csrValC** is zero, only sparisty pattern of **C** is calculated.

The functions **cusparseXcsrgeam2Nnz()** and **cusparse<t>csrgeam2()** supports the following properties if **pBuffer != NULL**

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	number of rows of sparse matrix **A**, **D** and **C**.
n	number of columns of sparse matrix **B**, **D** and **C**.
k	number of columns/rows of sparse matrix **A** / **B**.
alpha	<type> scalar used for multiplication.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzA	number of nonzero elements of sparse matrix **A**.
csrValA	<type> array of **nnzA** nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnzA** column indices of the nonzero elements of matrix **A**.
descrB	the descriptor of matrix **B**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzB	number of nonzero elements of sparse matrix **B**.
csrValB	<type> array of **nnzB** nonzero elements of matrix **B**.
csrRowPtrB	integer array of **k+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndB	integer array of **nnzB** column indices of the nonzero elements of matrix **B**.
descrD	the descriptor of matrix **D**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
nnzD	number of nonzero elements of sparse matrix **D**.
csrValD	<type> array of **nnzD** nonzero elements of matrix **D**.
csrRowPtrD	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndD	integer array of **nnzD** column indices of the nonzero elements of matrix **D**.

beta	<type> scalar used for multiplication.
descrC	the descriptor of matrix C. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL** only.
info	structure with information used in **csrgemm2Nnz** and **csrgemm2**.
pBuffer	buffer allocated by the user; the size is returned by **csrgemm2_bufferSizeExt**.

Output

csrValC	<type> array of **nnzC** nonzero elements of matrix C.
csrRowPtrC	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	integer array of **nnzC** column indices of the nonzero elements of matrix C.
pBufferSizeInBytes	number of bytes of the buffer used in **csrgemm2Nnnz** and **csrgemm2**.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory. It is equal to **(csrRowPtrC(m)-csrRowPtrC(0))**.

See `cusparseStatus_t` for the description of the return status

Chapter 11. CUSPARSE PRECONDITIONERS REFERENCE

This chapter describes the routines that implement different preconditioners.

11.1. Incomplete Cholesky Factorization: level 0

Different algorithms for ic0 are discussed in this section.

11.1.1. cusparse<t>csric02_bufferSize()

```
cusparseStatus_t
cusparseScsric02_bufferSize(cusparseHandle_t         handle,
                            int                      m,
                            int                      nnz,
                            const cusparseMatDescr_t descrA,
                            float*                   csrValA,
                            const int*               csrRowPtrA,
                            const int*               csrColIndA,
                            csric02Info_t            info,
                            int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsric02_bufferSize(cusparseHandle_t         handle,
                            int                      m,
                            int                      nnz,
                            const cusparseMatDescr_t descrA,
                            double*                  csrValA,
                            const int*               csrRowPtrA,
                            const int*               csrColIndA,
                            csric02Info_t            info,
                            int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseCcsric02_bufferSize(cusparseHandle_t         handle,
                            int                      m,
                            int                      nnz,
                            const cusparseMatDescr_t descrA,
                            cuComplex*               csrValA,
                            const int*               csrRowPtrA,
                            const int*               csrColIndA,
                            csric02Info_t            info,
                            int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseZcsric02_bufferSize(cusparseHandle_t         handle,
                            int                      m,
                            int                      nnz,
                            const cusparseMatDescr_t descrA,
                            cuDoubleComplex*         csrValA,
                            const int*               csrRowPtrA,
                            const int*               csrColIndA,
                            csric02Info_t            info,
                            int*                     pBufferSizeInBytes)
```

This function returns size of buffer used in computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

$$A \approx LL^H$$

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**.

The buffer size depends on dimension **m** and **nnz**, the number of nonzeros of the matrix. If the user changes the matrix, it is necessary to call **csric02_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	number of rows and columns of matrix **A**.
nnz	number of nonzeros of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** ($=$ **csrRowPtrA(m)** $-$ **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** $+1$ elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** ($=$ **csrRowPtrA(m)** $-$ **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

Output

info	record internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in **csric02_analysis()** and **csric02()**.

See `cusparseStatus_t` for the description of the return status

11.1.2. cusparse<t>csric02_analysis()

```
cusparseStatus_t
cusparseScsric02_analysis(cusparseHandle_t         handle,
                          int                      m,
                          int                      nnz,
                          const cusparseMatDescr_t descrA,
                          const float*             csrValA,
                          const int*               csrRowPtrA,
                          const int*               csrColIndA,
                          csric02Info_t            info,
                          cusparseSolvePolicy_t    policy,
                          void*                    pBuffer)

cusparseStatus_t
cusparseDcsric02_analysis(cusparseHandle_t         handle,
                          int                      m,
                          int                      nnz,
                          const cusparseMatDescr_t descrA,
                          const double*            csrValA,
                          const int*               csrRowPtrA,
                          const int*               csrColIndA,
                          csric02Info_t            info,
                          cusparseSolvePolicy_t    policy,
                          void*                    pBuffer)

cusparseStatus_t
cusparseCcsric02_analysis(cusparseHandle_t         handle,
                          int                      m,
                          int                      nnz,
                          const cusparseMatDescr_t descrA,
                          const cuComplex*         csrValA,
                          const int*               csrRowPtrA,
                          const int*               csrColIndA,
                          csric02Info_t            info,
                          cusparseSolvePolicy_t    policy,
                          void*                    pBuffer)

cusparseStatus_t
cusparseZcsric02_analysis(cusparseHandle_t         handle,
                          int                      m,
                          int                      nnz,
                          const cusparseMatDescr_t descrA,
                          const cuDoubleComplex*   csrValA,
                          const int*               csrRowPtrA,
                          const int*               csrColIndA,
                          csric02Info_t            info,
                          cusparseSolvePolicy_t    policy,
                          void*                    pBuffer)
```

This function performs the analysis phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

$$A \approx LL^H$$

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**.

This function requires a buffer size returned by **`csric02_bufferSize()`**. The address of **`pBuffer`** must be multiple of 128 bytes. If not, **`CUSPARSE_STATUS_INVALID_VALUE`** is returned.

Function **`csric02_analysis()`** reports a structural zero and computes level information stored in the opaque structure **`info`**. The level information can extract more parallelism during incomplete Cholesky factorization. However **`csric02()`** can be done without level information. To disable level information, the user must specify the policy of **`csric02_analysis()`** and **`csric02()`** as **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`**.

Function **`csric02_analysis()`** always reports the first structural zero, even if the policy is **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`**. The user needs to call **`cusparseXcsric02_zeroPivot()`** to know where the structural zero is.

It is the user's choice whether to call **`csric02()`** if **`csric02_analysis()`** reports a structural zero. In this case, the user can still call **`csric02()`**, which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows and columns of matrix **`A`**.
`nnz`	number of nonzeros of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`**.
`csrValA`	<type> array of **`nnz`** ($=$ **`csrRowPtrA(m)`** $-$ **`csrRowPtrA(0)`**) nonzero elements of matrix **`A`**.
`csrRowPtrA`	integer array of **`m`** $+1$ elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of **`nnz`** ($=$ **`csrRowPtrA(m)`** $-$ **`csrRowPtrA(0)`**) column indices of the nonzero elements of matrix **`A`**.
`info`	structure initialized using **`cusparseCreateCsric02Info()`**.
`policy`	the supported policies are **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`** and **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**.
`pBuffer`	buffer allocated by the user; the size is returned by **`csric02_bufferSize()`**.

Output

info	number of bytes of the buffer used in **csric02_analysis()** and **csric02()**.

See cusparseStatus_t for the description of the return status

11.1.3. cusparse<t>csric02()

```
cusparseStatus_t
cusparseScsric02(cusparseHandle_t         handle,
                 int                      m,
                 int                      nnz,
                 const cusparseMatDescr_t descrA,
                 float*                   csrValA_valM,
                 const int*               csrRowPtrA,
                 const int*               csrColIndA,
                 csric02Info_t            info,
                 cusparseSolvePolicy_t    policy,
                 void*                    pBuffer)

cusparseStatus_t
cusparseDcsric02(cusparseHandle_t         handle,
                 int                      m,
                 int                      nnz,
                 const cusparseMatDescr_t descrA,
                 double*                  csrValA_valM,
                 const int*               csrRowPtrA,
                 const int*               csrColIndA,
                 csric02Info_t            info,
                 cusparseSolvePolicy_t    policy,
                 void*                    pBuffer)

cusparseStatus_t
cusparseCcsric02(cusparseHandle_t         handle,
                 int                      m,
                 int                      nnz,
                 const cusparseMatDescr_t descrA,
                 cuComplex*               csrValA_valM,
                 const int*               csrRowPtrA,
                 const int*               csrColIndA,
                 csric02Info_t            info,
                 cusparseSolvePolicy_t    policy,
                 void*                    pBuffer)

cusparseStatus_t
cusparseZcsric02(cusparseHandle_t         handle,
                 int                      m,
                 int                      nnz,
                 const cusparseMatDescr_t descrA,
                 cuDoubleComplex*         csrValA_valM,
                 const int*               csrRowPtrA,
                 const int*               csrColIndA,
                 csric02Info_t            info,
                 cusparseSolvePolicy_t    policy,
                 void*                    pBuffer)
```

This function performs the solve phase of the computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting:

$$A \approx LL^H$$

This function requires a buffer size returned by **`csric02_bufferSize()`**. The address of **`pBuffer`** must be a multiple of 128 bytes. If not, **`CUSPARSE_STATUS_INVALID_VALUE`** is returned.

Although **`csric02()`** can be done without level information, the user still needs to be aware of consistency. If **`csric02_analysis()`** is called with policy **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**, **`csric02()`** can be run with or without levels. On the other hand, if **`csric02_analysis()`** is called with **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`**, **`csric02()`** can only accept **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`**; otherwise, **`CUSPARSE_STATUS_INVALID_VALUE`** is returned.

Function **`csric02()`** reports the first numerical zero, including a structural zero. The user must call **`cusparseXcsric02_zeroPivot()`** to know where the numerical zero is.

Function **`csric02()`** only takes the lower triangular part of matrix **`A`** to perform factorization. The matrix type must be **`CUSPARSE_MATRIX_TYPE_GENERAL`**, the fill mode and diagonal type are ignored, and the strictly upper triangular part is ignored and never touched. It does not matter if **`A`** is Hermitian or not. In other words, from the point of view of **`csric02()`** **`A`** is Hermitian and only the lower triangular part is provided.

> In practice, a positive definite matrix may not have incomplete cholesky factorization. To the best of our knowledge, only matrix **`M`** can guarantee the existence of incomplete cholesky factorization. If **`csric02()`** failed cholesky factorization and reported a numerical zero, it is possible that incomplete cholesky factorization does not exist.

For example, suppose **`A`** is a real **`m × m`** matrix, the following code solves the precondition system **`M*y = x`** where **`M`** is the product of Cholesky factorization **`L`** and its transpose.

$$M = LL^H$$

```
// Suppose that A is m x m sparse matrix represented by CSR format,
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_csrRowPtr, d_csrColInd, d_csrVal) is CSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.

cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
csric02Info_t info_M  = 0;
csrsv2Info_t  info_L  = 0;
csrsv2Info_t  info_Lt = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_Lt;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_Lt = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_Lt = CUSPARSE_OPERATION_TRANSPOSE;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for csric02 and two info's for csrsv2
cusparseCreateCsric02Info(&info_M);
cusparseCreateCsrsv2Info(&info_L);
cusparseCreateCsrsv2Info(&info_Lt);

// step 3: query how much memory used in csric02 and csrsv2, and allocate the
 buffer
cusparseDcsric02_bufferSize(handle, m, nnz,
    descr_M, d_csrVal, d_csrRowPtr, d_csrColInd, info_M, &bufferSize_M);
cusparseDcsrsv2_bufferSize(handle, trans_L, m, nnz,
    descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_L, &pBufferSize_L);
cusparseDcsrsv2_bufferSize(handle, trans_Lt, m, nnz,
    descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_Lt,&pBufferSize_Lt);

pBufferSize = max(bufferSize_M, max(pBufferSize_L, pBufferSize_Lt));

// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis of incomplete Cholesky on M
//         perform analysis of triangular solve on L
//         perform analysis of triangular solve on L'
// The lower triangular part of M has the same sparsity pattern as L, so
// we can do analysis of csric02 and csrsv2 simultaneously.

cusparseDcsric02_analysis(handle, m, nnz, descr_M,
    d_csrVal, d_csrRowPtr, d_csrColInd, info_M,
    policy_M, pBuffer);
status = cusparseXcsric02_zeroPivot(handle, info_M, &structural_zero);
```

The function supports the following properties if **`pBuffer != NULL`**

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows and columns of matrix **`A`**.
`nnz`	number of nonzeros of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`**.
`csrValA_valM`	<type> array of **`nnz`** (= **`csrRowPtrA(m)`** - **`csrRowPtrA(0)`**) nonzero elements of matrix **`A`**.
`csrRowPtrA`	integer array of **`m`** $+1$ elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of **`nnz`** (= **`csrRowPtrA(m)`** - **`csrRowPtrA(0)`**) column indices of the nonzero elements of matrix **`A`**.
`info`	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
`policy`	the supported policies are **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`** and **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**.
`pBuffer`	buffer allocated by the user; the size is returned by **`csric02_bufferSize()`**.

Output

`csrValA_valM`	<type> matrix containing the incomplete-Cholesky lower triangular factor.

See `cusparseStatus_t` for the description of the return status

11.1.4. cusparseXcsric02_zeroPivot()

```
cusparseStatus_t
cusparseXcsric02_zeroPivot(cusparseHandle_t handle,
                           csric02Info_t    info,
                           int*             position)
```

If the returned error code is **`CUSPARSE_STATUS_ZERO_PIVOT`**, **`position=j`** means **`A(j,j)`** has either a structural zero or a numerical zero; otherwise, **`position=-1`**.

The **`position`** can be 0-based or 1-based, the same as the matrix.

Function **`cusparseXcsric02_zeroPivot()`** is a blocking call. It calls **`cudaDeviceSynchronize()`** to make sure all previous kernels are done.

The **`position`** can be in the host memory or device memory. The user can set proper mode with **`cusparseSetPointerMode()`**.

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`info`	**`info`** contains structural zero or numerical zero if the user already called **`csric02_analysis()`** or **`csric02()`**.

Output

`position`	if no structural or numerical zero, **`position`** is -1; otherwise, if **`A(j,j)`** is missing or **`L(j,j)`** is zero, **`position=j`**.

See `cusparseStatus_t` for the description of the return status

11.1.5. cusparse<t>bsric02_bufferSize()

```
cusparseStatus_t
cusparseSbsric02_bufferSize(cusparseHandle_t         handle,
                            cusparseDirection_t      dirA,
                            int                      mb,
                            int                      nnzb,
                            const cusparseMatDescr_t descrA,
                            float*                   bsrValA,
                            const int*               bsrRowPtrA,
                            const int*               bsrColIndA,
                            int                      blockDim,
                            bsric02Info_t            info,
                            int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseDbsric02_bufferSize(cusparseHandle_t         handle,
                            cusparseDirection_t      dirA,
                            int                      mb,
                            int                      nnzb,
                            const cusparseMatDescr_t descrA,
                            double*                  bsrValA,
                            const int*               bsrRowPtrA,
                            const int*               bsrColIndA,
                            int                      blockDim,
                            bsric02Info_t            info,
                            int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseCbsric02_bufferSize(cusparseHandle_t         handle,
                            cusparseDirection_t      dirA,
                            int                      mb,
                            int                      nnzb,
                            const cusparseMatDescr_t descrA,
                            cuComplex*               bsrValA,
                            const int*               bsrRowPtrA,
                            const int*               bsrColIndA,
                            int                      blockDim,
                            bsric02Info_t            info,
                            int*                     pBufferSizeInBytes)

cusparseStatus_t
cusparseZbsric02_bufferSize(cusparseHandle_t         handle,
                            cusparseDirection_t      dirA,
                            int                      mb,
                            int                      nnzb,
                            const cusparseMatDescr_t descrA,
                            cuDoubleComplex*         bsrValA,
                            const int*               bsrRowPtrA,
                            const int*               bsrColIndA,
                            int                      blockDim,
                            bsric02Info_t            info,
                            int*                     pBufferSizeInBytes)
```

This function returns the size of a buffer used in computing the incomplete-Cholesky factorization with 0 fill-in and no pivoting

$$A \approx LL^H$$

A is an **(mb*blockDim)*(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**.

The buffer size depends on the dimensions of **mb**, **blockDim**, and the number of nonzero blocks of the matrix **nnzb**. If the user changes the matrix, it is necessary to call **bsric02_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and block columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** ($=$ **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** ($=$ **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)**) column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.

Output

info	record internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in **bsric02_analysis()** and **bsric02()**.

See `cusparseStatus_t` for the description of the return status

11.1.6. cusparse<t>bsric02_analysis()

```
cusparseStatus_t
cusparseSbsric02_analysis(cusparseHandle_t        handle,
                          cusparseDirection_t     dirA,
                          int                      mb,
                          int                      nnzb,
                          const cusparseMatDescr_t descrA,
                          const float*             bsrValA,
                          const int*               bsrRowPtrA,
                          const int*               bsrColIndA,
                          int                      blockDim,
                          bsric02Info_t            info,
                          cusparseSolvePolicy_t    policy,
                          void*                    pBuffer)

cusparseStatus_t
cusparseDbsric02_analysis(cusparseHandle_t        handle,
                          cusparseDirection_t     dirA,
                          int                      mb,
                          int                      nnzb,
                          const cusparseMatDescr_t descrA,
                          const double*            bsrValA,
                          const int*               bsrRowPtrA,
                          const int*               bsrColIndA,
                          int                      blockDim,
                          bsric02Info_t            info,
                          cusparseSolvePolicy_t    policy,
                          void*                    pBuffer)

cusparseStatus_t
cusparseCbsric02_analysis(cusparseHandle_t        handle,
                          cusparseDirection_t     dirA,
                          int                      mb,
                          int                      nnzb,
                          const cusparseMatDescr_t descrA,
                          const cuComplex*         bsrValA,
                          const int*               bsrRowPtrA,
                          const int*               bsrColIndA,
                          int                      blockDim,
                          bsric02Info_t            info,
                          cusparseSolvePolicy_t    policy,
                          void*                    pBuffer)

cusparseStatus_t
cusparseZbsric02_analysis(cusparseHandle_t        handle,
                          cusparseDirection_t     dirA,
                          int                      mb,
                          int                      nnzb,
                          const cusparseMatDescr_t descrA,
                          const cuDoubleComplex*   bsrValA,
                          const int*               bsrRowPtrA,
                          const int*               bsrColIndA,
                          int                      blockDim,
                          bsric02Info_t            info,
                          cusparseSolvePolicy_t    policy,
                          void*                    pBuffer)
```

This function performs the analysis phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting

$$A \approx LL^H$$

A is an **(mb*blockDim)x(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**. The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by **bsric02_bufferSize90**. The address of **pBuffer** must be a multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function**bsric02_analysis()** reports structural zero and computes level information stored in the opaque structure **info**. The level information can extract more parallelism during incomplete Cholesky factorization. However **bsric02()** can be done without level information. To disable level information, the user needs to specify the parameter **policy** of **bsric02[_analysis|]** as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **bsric02_analysis** always reports the first structural zero, even when parameter **policy** is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. The user must call **cusparseXbsric02_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **bsric02()** if **bsric02_analysis()** reports a structural zero. In this case, the user can still call **bsric02()**, which returns a numerical zero in the same position as the structural zero. However the result is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and block columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** - **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.

bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** $($= **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A; must be larger than zero.
info	structure initialized using **cusparseCreateBsric02Info()**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user; the size is returned by **bsric02_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

See `cusparseStatus_t` for the description of the return status

11.1.7. cusparse<t>bsric02()

```
cusparseStatus_t
cusparseSbsric02(cusparseHandle_t         handle,
                 cusparseDirection_t      dirA,
                 int                      mb,
                 int                      nnzb,
                 const cusparseMatDescr_t descrA,
                 float*                   bsrValA,
                 const int*               bsrRowPtrA,
                 const int*               bsrColIndA,
                 int                      blockDim,
                 bsric02Info_t            info,
                 cusparseSolvePolicy_t    policy,
                 void*                    pBuffer)

cusparseStatus_t
cusparseDbsric02(cusparseHandle_t         handle,
                 cusparseDirection_t      dirA,
                 int                      mb,
                 int                      nnzb,
                 const cusparseMatDescr_t descrA,
                 double*                  bsrValA,
                 const int*               bsrRowPtrA,
                 const int*               bsrColIndA,
                 int                      blockDim,
                 bsric02Info_t            info,
                 cusparseSolvePolicy_t    policy,
                 void*                    pBuffer)

cusparseStatus_t
cusparseCbsric02(cusparseHandle_t         handle,
                 cusparseDirection_t      dirA,
                 int                      mb,
                 int                      nnzb,
                 const cusparseMatDescr_t descrA,
                 cuComplex*               bsrValA,
                 const int*               bsrRowPtrA,
                 const int*               bsrColIndA,
                 int                      blockDim,
                 bsric02Info_t            info,
                 cusparseSolvePolicy_t    policy,
                 void*                    pBuffer)

cusparseStatus_t
cusparseZbsric02(cusparseHandle_t         handle,
                 cusparseDirection_t      dirA,
                 int                      mb,
                 int                      nnzb,
                 const cusparseMatDescr_t descrA,
                 cuDoubleComplex*         bsrValA,
                 const int*               bsrRowPtrA,
                 const int*               bsrColIndA,
                 int                      blockDim,
                 bsric02Info_t            info,
                 cusparseSolvePolicy_t    policy,
                 void*                    pBuffer)
```

This function performs the solve phase of the incomplete-Cholesky factorization with 0 fill-in and no pivoting

$$A \approx LL^H$$

A is an **(mb*blockDim)×(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**. The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by **bsric02_bufferSize()**. The address of **pBuffer** must be a multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **bsric02()** can be done without level information, the user must be aware of consistency. If **bsric02_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **bsric02()** can be run with or without levels. On the other hand, if **bsric02_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **bsric02()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsric02()** has the same behavior as **csric02()**. That is, **bsr2csr(bsric02(A)) = csric02(bsr2csr(A))**. The numerical zero of **csric02()** means there exists some zero **L(j,j)**. The numerical zero of **bsric02()** means there exists some block **Lj,j)** that is not invertible.

Function **bsric02** reports the first numerical zero, including a structural zero. The user must call **cusparseXbsric02_zeroPivot()** to know where the numerical zero is.

The **bsric02()** function only takes the lower triangular part of matrix **A** to perform factorization. The strictly upper triangular part is ignored and never touched. It does not matter if **A** is Hermitian or not. In other words, from the point of view of **bsric02()**, **A** is Hermitian and only the lower triangular part is provided. Moreover, the imaginary part of diagonal elements of diagonal blocks is ignored.

For example, suppose **A** is a real m-by-m matrix, where **m=mb*blockDim**. The following code solves precondition system **M*y = x**, where **M** is the product of Cholesky factorization **L** and its transpose.

$$M = LL^H$$

```
// Suppose that A is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.
// - d_x, d_y and d_z are of size m.
cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
bsric02Info_t info_M  = 0;
bsrsv2Info_t  info_L  = 0;
bsrsv2Info_t  info_Lt = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_Lt;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L  = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_Lt = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_Lt = CUSPARSE_OPERATION_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for bsric02 and two info's for bsrsv2
cusparseCreateBsric02Info(&info_M);
cusparseCreateBsrsv2Info(&info_L);
cusparseCreateBsrsv2Info(&info_Lt);

// step 3: query how much memory used in bsric02 and bsrsv2, and allocate the
 buffer
cusparseDbsric02_bufferSize(handle, dir, mb, nnzb,
    descr_M, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_M,
 &bufferSize_M);
cusparseDbsrsv2_bufferSize(handle, dir, trans_L, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L,
 &pBufferSize_L);
cusparseDbsrsv2_bufferSize(handle, dir, trans_Lt, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_Lt,
 &pBufferSize_Lt);

pBufferSize = max(bufferSize_M, max(pBufferSize_L, pBufferSize_Lt));

// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis of incomplete Cholesky on M
//         perform analysis of triangular solve on L
//         perform analysis of triangular solve on L'
```

The function supports the following properties if **`pBuffer != NULL`**

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`dirA`	storage format of blocks, either **`CUSPARSE_DIRECTION_ROW`** or **`CUSPARSE_DIRECTION_COLUMN`**.
`mb`	number of block rows and block columns of matrix **`A`**.
`nnzb`	number of nonzero blocks of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`**.
`bsrValA`	<type> array of **`nnzb`** $($= **`bsrRowPtrA(mb)`** $-$ **`bsrRowPtrA(0)`** $)$ nonzero blocks of matrix **`A`**.
`bsrRowPtrA`	integer array of **`mb`** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
`bsrColIndA`	integer array of **`nnzb`** $($= **`bsrRowPtrA(mb)`** $-$ **`bsrRowPtrA(0)`** $)$ column indices of the nonzero blocks of matrix **`A`**.
`blockDim`	block dimension of sparse matrix A, larger than zero.
`info`	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
`policy`	the supported policies are **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`** and **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**.
`pBuffer`	buffer allocated by the user, the size is returned by **`bsric02_bufferSize()`**.

Output

`bsrValA`	<type> matrix containing the incomplete-Cholesky lower triangular factor.

See `cusparseStatus_t` for the description of the return status

11.1.8. cusparseXbsric02_zeroPivot()

```
cusparseStatus_t
cusparseXbsric02_zeroPivot(cusparseHandle_t handle,
                           bsric02Info_t    info,
                           int*             position)
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero (the block is not positive definite). Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXbsric02_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
info	**info** contains a structural zero or a numerical zero if the user already called **bsric02_analysis()** or **bsric02()**.

Output

position	if no structural or numerical zero, **position** is -1, otherwise if **A(j,j)** is missing or **L(j,j)** is not positive definite, **position=j**.

See `cusparseStatus_t` for the description of the return status

11.2. Incomplete LU Factorization: level 0

Different algorithms for ilu0 are discussed in this section.

11.2.1. cusparse<t>csrilu02_numericBoost()

```
cusparseStatus_t
cusparseScsrilu02_numericBoost(cusparseHandle_t handle,
                               csrilu02Info_t   info,
                               int              enable_boost,
                               double*          tol,
                               float*           boost_val)

cusparseStatus_t
cusparseDcsrilu02_numericBoost(cusparseHandle_t handle,
                               csrilu02Info_t   info,
                               int              enable_boost,
                               double*          tol,
                               double*          boost_val)

cusparseStatus_t
cusparseCcsrilu02_numericBoost(cusparseHandle_t handle,
                               csrilu02Info_t   info,
                               int              enable_boost,
                               double*          tol,
                               cuComplex*       boost_val)

cusparseStatus_t
cusparseZcsrilu02_numericBoost(cusparseHandle_t handle,
                               csrilu02Info_t   info,
                               int              enable_boost,
                               double*          tol,
                               cuDoubleComplex* boost_val)
```

The user can use a boost value to replace a numerical value in incomplete LU factorization. The **tol** is used to determine a numerical zero, and the **boost_val** is used to replace a numerical zero. The behavior is

if **tol >= fabs(A(j,j))**, then **A(j,j)=boost_val**.

To enable a boost value, the user has to set parameter **enable_boost** to 1 before calling **csrilu02()**. To disable a boost value, the user can call **csrilu02_numericBoost()** again with parameter **enable_boost=0**.

If **enable_boost=0**, **tol** and **boost_val** are ignored.

Both **tol** and **boost_val** can be in the host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
info	structure initialized using **cusparseCreateCsrilu02Info()**.
enable_boost	disable boost by **enable_boost=0**; otherwise, boost is enabled.

tol	tolerance to determine a numerical zero.
boost_val	boost value to replace a numerical zero.

See `cusparseStatus_t` for the description of the return status

11.2.2. cusparse<t>csrilu02_bufferSize()

```
cusparseStatus_t
cusparseScsrilu02_bufferSize(cusparseHandle_t         handle,
                             int                      m,
                             int                      nnz,
                             const cusparseMatDescr_t descrA,
                             float*                   csrValA,
                             const int*               csrRowPtrA,
                             const int*               csrColIndA,
                             csrilu02Info_t           info,
                             int*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsrilu02_bufferSize(cusparseHandle_t         handle,
                             int                      m,
                             int                      nnz,
                             const cusparseMatDescr_t descrA,
                             double*                  csrValA,
                             const int*               csrRowPtrA,
                             const int*               csrColIndA,
                             csrilu02Info_t           info,
                             int*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseCcsrilu02_bufferSize(cusparseHandle_t         handle,
                             int                      m,
                             int                      nnz,
                             const cusparseMatDescr_t descrA,
                             cuComplex*               csrValA,
                             const int*               csrRowPtrA,
                             const int*               csrColIndA,
                             csrilu02Info_t           info,
                             int*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseZcsrilu02_bufferSize(cusparseHandle_t         handle,
                             int                      m,
                             int                      nnz,
                             const cusparseMatDescr_t descrA,
                             cuDoubleComplex*         csrValA,
                             const int*               csrRowPtrA,
                             const int*               csrColIndA,
                             csrilu02Info_t           info,
                             int*
 pBufferSizeInBytes)
```

This function returns size of the buffer used in computing the incomplete-LU factorization with 0 fill-in and no pivoting:

$$A \approx LU$$

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**.

The buffer size depends on the dimension **m** and **nnz**, the number of nonzeros of the matrix. If the user changes the matrix, it is necessary to call **csrilu02_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	number of rows and columns of matrix **A**.
nnz	number of nonzeros of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m** $+1$ elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** (= **csrRowPtrA(m)** - **csrRowPtrA(0)**) column indices of the nonzero elements of matrix **A**.

Output

info	record internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in **csrilu02_analysis()** and **csrilu02()**.

See `cusparseStatus_t` for the description of the return status

11.2.3. cusparse<t>csrilu02_analysis()

```
cusparseStatus_t
cusparseScsrilu02_analysis(cusparseHandle_t         handle,
                           int                      m,
                           int                      nnz,
                           const cusparseMatDescr_t descrA,
                           const float*             csrValA,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           csrilu02Info_t           info,
                           cusparseSolvePolicy_t    policy,
                           void*                    pBuffer)

cusparseStatus_t
cusparseDcsrilu02_analysis(cusparseHandle_t         handle,
                           int                      m,
                           int                      nnz,
                           const cusparseMatDescr_t descrA,
                           const double*            csrValA,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           csrilu02Info_t           info,
                           cusparseSolvePolicy_t    policy,
                           void*                    pBuffer)

cusparseStatus_t
cusparseCcsrilu02_analysis(cusparseHandle_t         handle,
                           int                      m,
                           int                      nnz,
                           const cusparseMatDescr_t descrA,
                           const cuComplex*         csrValA,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           csrilu02Info_t           info,
                           cusparseSolvePolicy_t    policy,
                           void*                    pBuffer)

cusparseStatus_t
cusparseZcsrilu02_analysis(cusparseHandle_t         handle,
                           int                      m,
                           int                      nnz,
                           const cusparseMatDescr_t descrA,
                           const cuDoubleComplex*   csrValA,
                           const int*               csrRowPtrA,
                           const int*               csrColIndA,
                           csrilu02Info_t           info,
                           cusparseSolvePolicy_t    policy,
                           void*                    pBuffer)
```

This function performs the analysis phase of the incomplete-LU factorization with 0 fill-in and no pivoting:

$$A \approx LU$$

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**.

This function requires the buffer size returned by **`csrilu02_bufferSize()`**. The address of **`pBuffer`** must be a multiple of 128 bytes. If not, **`CUSPARSE_STATUS_INVALID_VALUE`** is returned.

Function **`csrilu02_analysis()`** reports a structural zero and computes level information stored in the opaque structure **`info`**. The level information can extract more parallelism during incomplete LU factorization; however **`csrilu02()`** can be done without level information. To disable level information, the user must specify the policy of **`csrilu02()`** as **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`**.

It is the user's choice whether to call **`csrilu02()`** if **`csrilu02_analysis()`** reports a structural zero. In this case, the user can still call **`csrilu02()`**, which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows and columns of matrix **`A`**.
`nnz`	number of nonzeros of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`**.
`csrValA`	<type> array of **`nnz`** (= **`csrRowPtrA(m)`** − **`csrRowPtrA(0)`**) nonzero elements of matrix **`A`**.
`csrRowPtrA`	integer array of **`m`** $+1$ elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of **`nnz`** (= **`csrRowPtrA(m)`** − **`csrRowPtrA(0)`**) column indices of the nonzero elements of matrix **`A`**.
`info`	structure initialized using **`cusparseCreateCsrilu02Info()`**.
`policy`	the supported policies are **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`** and **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**.
`pBuffer`	buffer allocated by the user, the size is returned by **`csrilu02_bufferSize()`**.

Output

`info`	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

See cusparseStatus_t for the description of the return status

11.2.4. cusparse<t>csrilu02()

```
cusparseStatus_t
cusparseScsrilu02(cusparseHandle_t         handle,
                  int                      m,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  float*                   csrValA_valM,
                  const int*               csrRowPtrA,
                  const int*               csrColIndA,
                  csrilu02Info_t           info,
                  cusparseSolvePolicy_t    policy,
                  void*                    pBuffer)

cusparseStatus_t
cusparseDcsrilu02(cusparseHandle_t         handle,
                  int                      m,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  double*                  csrValA_valM,
                  const int*               csrRowPtrA,
                  const int*               csrColIndA,
                  csrilu02Info_t           info,
                  cusparseSolvePolicy_t    policy,
                  void*                    pBuffer)

cusparseStatus_t
cusparseCcsrilu02(cusparseHandle_t         handle,
                  int                      m,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  cuComplex*               csrValA_valM,
                  const int*               csrRowPtrA,
                  const int*               csrColIndA,
                  csrilu02Info_t           info,
                  cusparseSolvePolicy_t    policy,
                  void*                    pBuffer)

cusparseStatus_t
cusparseZcsrilu02(cusparseHandle_t         handle,
                  int                      m,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  cuDoubleComplex*         csrValA_valM,
                  const int*               csrRowPtrA,
                  const int*               csrColIndA,
                  csrilu02Info_t           info,
                  cusparseSolvePolicy_t    policy,
                  void*                    pBuffer)
```

This function performs the solve phase of the incomplete-LU factorization with 0 fill-in and no pivoting:

$$A \approx LU$$

A is an **m×m** sparse matrix that is defined in CSR storage format by the three arrays **csrValA_valM**, **csrRowPtrA**, and **csrColIndA**.

This function requires a buffer size returned by **csrilu02_bufferSize()**. The address of **pBuffer** must be a multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**. The fill mode and diagonal type are ignored.

Although **csrilu02()** can be done without level information, the user still needs to be aware of consistency. If **csrilu02_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **csrilu02()** can be run with or without levels. On the other hand, if **csrilu02_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **csrilu02()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **csrilu02()** reports the first numerical zero, including a structural zero. The user must call **cusparseXcsrilu02_zeroPivot()** to know where the numerical zero is.

For example, suppose **A** is a real **m × m** matrix, the following code solves precondition system **M*y = x** where **M** is the product of LU factors **L** and **U**.

```
// Suppose that A is m x m sparse matrix represented by CSR format,
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_csrRowPtr, d_csrColInd, d_csrVal) is CSR of A on device memory,
// - d_x is right hand side vector on device memory,
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.

cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
cusparseMatDescr_t descr_U = 0;
csrilu02Info_t info_M  = 0;
csrsv2Info_t  info_L  = 0;
csrsv2Info_t  info_U  = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_U;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_U = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_U  = CUSPARSE_OPERATION_NON_TRANSPOSE;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal
// - matrix U is base-1
// - matrix U is upper triangular
// - matrix U has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_UNIT);

cusparseCreateMatDescr(&descr_U);
cusparseSetMatIndexBase(descr_U, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_U, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_U, CUSPARSE_FILL_MODE_UPPER);
cusparseSetMatDiagType(descr_U, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for csrilu02 and two info's for csrsv2
cusparseCreateCsrilu02Info(&info_M);
cusparseCreateCsrsv2Info(&info_L);
cusparseCreateCsrsv2Info(&info_U);

// step 3: query how much memory used in csrilu02 and csrsv2, and allocate the
 buffer
cusparseDcsrilu02_bufferSize(handle, m, nnz,
    descr_M, d_csrVal, d_csrRowPtr, d_csrColInd, info_M, &pBufferSize_M);
cusparseDcsrsv2_bufferSize(handle, trans_L, m, nnz,
    descr_L, d_csrVal, d_csrRowPtr, d_csrColInd, info_L, &pBufferSize_L);
cusparseDcsrsv2_bufferSize(handle, trans_U, m, nnz,
    descr_U, d_csrVal, d_csrRowPtr, d_csrColInd, info_U, &pBufferSize_U);

pBufferSize = max(pBufferSize_M, max(pBufferSize_L, pBufferSize_U));

// pBuffer returned by cudaMalloc is automatically aligned to 128 bytes.
cudaMalloc((void**)&pBuffer, pBufferSize);

// step 4: perform analysis of incomplete Cholesky on M
```

The function supports the following properties if **`pBuffer != NULL`**

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows and columns of matrix **`A`**.
`nnz`	number of nonzeros of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`**.
`csrValA_valM`	<type> array of **`nnz`** ($=$ **`csrRowPtrA(m)`** $-$ **`csrRowPtrA(0)`**) nonzero elements of matrix **`A`**.
`csrRowPtrA`	integer array of **`m`** $+1$ elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of **`nnz`** ($=$ **`csrRowPtrA(m)`** $-$ **`csrRowPtrA(0)`**) column indices of the nonzero elements of matrix **`A`**.
`info`	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
`policy`	the supported policies are **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`** and **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**.
`pBuffer`	buffer allocated by the user; the size is returned by **`csrilu02_bufferSize()`**.

Output

`csrValA_valM`	<type> matrix containing the incomplete-LU lower and upper triangular factors.

See `cusparseStatus_t` for the description of the return status

11.2.5. cusparseXcsrilu02_zeroPivot()

```
cusparseStatus_t
cusparseXcsrilu02_zeroPivot(cusparseHandle_t handle,
                            csrilu02Info_t   info,
                            int*             position)
```

If the returned error code is **`CUSPARSE_STATUS_ZERO_PIVOT`**, **`position=j`** means **`A(j,j)`** has either a structural zero or a numerical zero; otherwise, **`position=-1`**.

The **`position`** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXcsrilu02_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set proper mode with **cusparseSetPointerMode()**.

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **csrilu02_analysis()** or **csrilu02()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise if **A(j,j)** is missing or **U(j,j)** is zero, **position=j**.

See cusparseStatus_t for the description of the return status

11.2.6. cusparse<t>bsrilu02_numericBoost()

```
cusparseStatus_t
cusparseSbsrilu02_numericBoost(cusparseHandle_t handle,
                               bsrilu02Info_t   info,
                               int              enable_boost,
                               double*          tol,
                               float*           boost_val)

cusparseStatus_t
cusparseDbsrilu02_numericBoost(cusparseHandle_t handle,
                               bsrilu02Info_t   info,
                               int              enable_boost,
                               double*          tol,
                               double*          boost_val)

cusparseStatus_t
cusparseCbsrilu02_numericBoost(cusparseHandle_t handle,
                               bsrilu02Info_t   info,
                               int              enable_boost,
                               double*          tol,
                               cuComplex*       boost_val)

cusparseStatus_t
cusparseZbsrilu02_numericBoost(cusparseHandle_t handle,
                               bsrilu02Info_t   info,
                               int              enable_boost,
                               double*          tol,
                               cuDoubleComplex* boost_val)
```

The user can use a boost value to replace a numerical value in incomplete LU factorization. Parameter **tol** is used to determine a numerical zero, and **boost_val** is used to replace a numerical zero. The behavior is as follows:

if **tol >= fabs(A(j,j))**, then reset each diagonal element of block **A(j,j)** by **boost_val**.

To enable a boost value, the user sets parameter **enable_boost** to 1 before calling **bsrilu02()**. To disable the boost value, the user can call **bsrilu02_numericBoost()** with parameter **enable_boost=0**.

If **enable_boost=0**, **tol** and **boost_val** are ignored.

Both **tol** and **boost_val** can be in host memory or device memory. The user can set the proper mode with **cusparseSetPointerMode()**.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
info	structure initialized using **cusparseCreateBsrilu02Info()**.
enable_boost	disable boost by setting **enable_boost=0**. Otherwise, boost is enabled.
tol	tolerance to determine a numerical zero.
boost_val	boost value to replace a numerical zero.

See cusparseStatus_t for the description of the return status

11.2.7. cusparse<t>bsrilu02_bufferSize()

```
cusparseStatus_t
cusparseSbsrilu02_bufferSize(cusparseHandle_t handle,
                             cusparseDirection_t dirA,
                             int mb,
                             int nnzb,
                             const cusparseMatDescr_t descrA,
                             float *bsrValA,
                             const int *bsrRowPtrA,
                             const int *bsrColIndA,
                             int blockDim,
                             bsrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseDbsrilu02_bufferSize(cusparseHandle_t handle,
                             cusparseDirection_t dirA,
                             int mb,
                             int nnzb,
                             const cusparseMatDescr_t descrA,
                             double *bsrValA,
                             const int *bsrRowPtrA,
                             const int *bsrColIndA,
                             int blockDim,
                             bsrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseCbsrilu02_bufferSize(cusparseHandle_t handle,
                             cusparseDirection_t dirA,
                             int mb,
                             int nnzb,
                             const cusparseMatDescr_t descrA,
                             cuComplex *bsrValA,
                             const int *bsrRowPtrA,
                             const int *bsrColIndA,
                             int blockDim,
                             bsrilu02Info_t info,
                             int *pBufferSizeInBytes);

cusparseStatus_t
cusparseZbsrilu02_bufferSize(cusparseHandle_t handle,
                             cusparseDirection_t dirA,
                             int mb,
                             int nnzb,
                             const cusparseMatDescr_t descrA,
                             cuDoubleComplex *bsrValA,
                             const int *bsrRowPtrA,
                             const int *bsrColIndA,
                             int blockDim,
                             bsrilu02Info_t info,
                             int *pBufferSizeInBytes);
```

This function returns the size of the buffer used in computing the incomplete-LU factorization with 0 fill-in and no pivoting

$$A \approx LU$$

A is an **(mb*blockDim)*(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**.

The buffer size depends on the dimensions of **mb**, **blockDim**, and the number of nonzero blocks of the matrix **nnzb**. If the user changes the matrix, it is necessary to call **bsrilu02_bufferSize()** again to have the correct buffer size; otherwise, a segmentation fault may occur.

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** $($= **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ nonzero blocks of matrix **A**.
bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** $($= **bsrRowPtrA(mb)** $-$ **bsrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.

Output

info	record internal states based on different algorithms.
pBufferSizeInBytes	number of bytes of the buffer used in **bsrilu02_analysis()** and **bsrilu02()**.

Status Returned

CUSPARSE_STATUS_SUCCESS	the operation completed successfully.
CUSPARSE_STATUS_NOT_INITIALIZED	the library was not initialized.
CUSPARSE_STATUS_ALLOC_FAILED	the resources could not be allocated.
CUSPARSE_STATUS_INVALID_VALUE	invalid parameters were passed (**mb,nnzb<=0**), base index is not 0 or 1.

CUSPARSE_STATUS_ARCH_MISMATCH	the device only supports compute capability 2.0 and above.
CUSPARSE_STATUS_INTERNAL_ERROR	an internal operation failed.
CUSPARSE_STATUS_MATRIX_TYPE_NOT_SUPPORTED	the matrix type is not supported.

11.2.8. cusparse<t>bsrilu02_analysis()

```
cusparseStatus_t
cusparseSbsrilu02_analysis(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           int                      mb,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           float*                   bsrValA,
                           const int*               bsrRowPtrA,
                           const int*               bsrColIndA,
                           int                      blockDim,
                           bsrilu02Info_t           info,
                           cusparseSolvePolicy_t    policy,
                           void*                    pBuffer)

cusparseStatus_t
cusparseDbsrilu02_analysis(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           int                      mb,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           double*                  bsrValA,
                           const int*               bsrRowPtrA,
                           const int*               bsrColIndA,
                           int                      blockDim,
                           bsrilu02Info_t           info,
                           cusparseSolvePolicy_t    policy,
                           void*                    pBuffer)

cusparseStatus_t
cusparseCbsrilu02_analysis(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           int                      mb,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           cuComplex*               bsrValA,
                           const int*               bsrRowPtrA,
                           const int*               bsrColIndA,
                           int                      blockDim,
                           bsrilu02Info_t           info,
                           cusparseSolvePolicy_t    policy,
                           void*                    pBuffer)

cusparseStatus_t
cusparseZbsrilu02_analysis(cusparseHandle_t         handle,
                           cusparseDirection_t      dirA,
                           int                      mb,
                           int                      nnzb,
                           const cusparseMatDescr_t descrA,
                           cuDoubleComplex*         bsrValA,
                           const int*               bsrRowPtrA,
                           const int*               bsrColIndA,
                           int                      blockDim,
                           bsrilu02Info_t           info,
                           cusparseSolvePolicy_t    policy,
                           void*                    pBuffer)
```

This function performs the analysis phase of the incomplete-LU factorization with 0 fill-in and no pivoting

$$A \approx LU$$

A is an **(mb*blockDim)×(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**. The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major as determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored.

This function requires a buffer size returned by **bsrilu02_bufferSize()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrilu02_analysis()** reports a structural zero and computes level information stored in the opaque structure **info**. The level information can extract more parallelism during incomplete LU factorization. However **bsrilu02()** can be done without level information. To disable level information, the user needs to specify the parameter **policy** of **bsrilu02[_analysis|]** as **CUSPARSE_SOLVE_POLICY_NO_LEVEL**.

Function **bsrilu02_analysis()** always reports the first structural zero, even with parameter **policy** is **CUSPARSE_SOLVE_POLICY_NO_LEVEL**. The user must call **cusparseXbsrilu02_zeroPivot()** to know where the structural zero is.

It is the user's choice whether to call **bsrilu02()** if **bsrilu02_analysis()** reports a structural zero. In this case, the user can still call **bsrilu02()**, which will return a numerical zero at the same position as the structural zero. However the result is meaningless.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dirA	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows and block columns of matrix **A**.
nnzb	number of nonzero blocks of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb** (= **bsrRowPtrA(mb)** − **bsrRowPtrA(0)**) nonzero blocks of matrix **A**.

bsrRowPtrA	integer array of **mb** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
bsrColIndA	integer array of **nnzb** $($= **bsrRowPtrA(mb)** − **bsrRowPtrA(0)** $)$ column indices of the nonzero blocks of matrix **A**.
blockDim	block dimension of sparse matrix A, larger than zero.
info	structure initialized using **cusparseCreateBsrilu02Info()**.
policy	the supported policies are **CUSPARSE_SOLVE_POLICY_NO_LEVEL** and **CUSPARSE_SOLVE_POLICY_USE_LEVEL**.
pBuffer	buffer allocated by the user, the size is returned by **bsrilu02_bufferSize()**.

Output

info	structure filled with information collected during the analysis phase (that should be passed to the solve phase unchanged).

See `cusparseStatus_t` for the description of the return status

11.2.9. cusparse<t>bsrilu02()

```
cusparseStatus_t
cusparseSbsrilu02(cusparseHandle_t         handle,
                  cusparseDirection_t      dirA,
                  int                      mb,
                  int                      nnzb,
                  const cusparseMatDescr_t descry,
                  float*                   bsrValA,
                  const int*               bsrRowPtrA,
                  const int*               bsrColIndA,
                  int                      blockDim,
                  bsrilu02Info_t           info,
                  cusparseSolvePolicy_t    policy,
                  void*                    pBuffer)

cusparseStatus_t
cusparseDbsrilu02(cusparseHandle_t         handle,
                  cusparseDirection_t      dirA,
                  int                      mb,
                  int                      nnzb,
                  const cusparseMatDescr_t descry,
                  double*                  bsrValA,
                  const int*               bsrRowPtrA,
                  const int*               bsrColIndA,
                  int                      blockDim,
                  bsrilu02Info_t           info,
                  cusparseSolvePolicy_t    policy,
                  void*                    pBuffer)

cusparseStatus_t
cusparseCbsrilu02(cusparseHandle_t         handle,
                  cusparseDirection_t      dirA,
                  int                      mb,
                  int                      nnzb,
                  const cusparseMatDescr_t descry,
                  cuComplex*               bsrValA,
                  const int*               bsrRowPtrA,
                  const int*               bsrColIndA,
                  int                      blockDim,
                  bsrilu02Info_t           info,
                  cusparseSolvePolicy_t    policy,
                  void*                    pBuffer)

cusparseStatus_t
cusparseZbsrilu02(cusparseHandle_t         handle,
                  cusparseDirection_t      dirA,
                  int                      mb,
                  int                      nnzb,
                  const cusparseMatDescr_t descry,
                  cuDoubleComplex*         bsrValA,
                  const int*               bsrRowPtrA,
                  const int*               bsrColIndA,
                  int                      blockDim,
                  bsrilu02Info_t           info,
                  cusparseSolvePolicy_t    policy,
                  void*                    pBuffer)
```

This function performs the solve phase of the incomplete-LU factorization with 0 fill-in and no pivoting

$$A \approx LU$$

A is an **(mb*blockDim)×(mb*blockDim)** sparse matrix that is defined in BSR storage format by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**. The block in BSR format is of size **blockDim*blockDim**, stored as column-major or row-major determined by parameter **dirA**, which is either **CUSPARSE_DIRECTION_COLUMN** or **CUSPARSE_DIRECTION_ROW**. The matrix type must be **CUSPARSE_MATRIX_TYPE_GENERAL**, and the fill mode and diagonal type are ignored. Function **bsrilu02()** supports an arbitrary **blockDim**.

This function requires a buffer size returned by **bsrilu02_bufferSize()**. The address of **pBuffer** must be a multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Although **bsrilu02()** can be used without level information, the user must be aware of consistency. If **bsrilu02_analysis()** is called with policy **CUSPARSE_SOLVE_POLICY_USE_LEVEL**, **bsrilu02()** can be run with or without levels. On the other hand, if **bsrilu02_analysis()** is called with **CUSPARSE_SOLVE_POLICY_NO_LEVEL**, **bsrilu02()** can only accept **CUSPARSE_SOLVE_POLICY_NO_LEVEL**; otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Function **bsrilu02()** has the same behavior as **csrilu02()**. That is, **bsr2csr(bsrilu02(A)) = csrilu02(bsr2csr(A))**. The numerical zero of **csrilu02()** means there exists some zero **U(j,j)**. The numerical zero of **bsrilu02()** means there exists some block **U(j,j)** that is not invertible.

Function **bsrilu02** reports the first numerical zero, including a structural zero. The user must call **cusparseXbsrilu02_zeroPivot()** to know where the numerical zero is.

For example, suppose **A** is a real m-by-m matrix where **m=mb*blockDim**. The following code solves precondition system **M*y = x**, where **M** is the product of LU factors **L** and **U**.

```
// Suppose that A is m x m sparse matrix represented by BSR format,
// The number of block rows/columns is mb, and
// the number of nonzero blocks is nnzb.
// Assumption:
// - handle is already created by cusparseCreate(),
// - (d_bsrRowPtr, d_bsrColInd, d_bsrVal) is BSR of A on device memory,
// - d_x is right hand side vector on device memory.
// - d_y is solution vector on device memory.
// - d_z is intermediate result on device memory.
// - d_x, d_y and d_z are of size m.
cusparseMatDescr_t descr_M = 0;
cusparseMatDescr_t descr_L = 0;
cusparseMatDescr_t descr_U = 0;
bsrilu02Info_t info_M = 0;
bsrsv2Info_t   info_L = 0;
bsrsv2Info_t   info_U = 0;
int pBufferSize_M;
int pBufferSize_L;
int pBufferSize_U;
int pBufferSize;
void *pBuffer = 0;
int structural_zero;
int numerical_zero;
const double alpha = 1.;
const cusparseSolvePolicy_t policy_M = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_L = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
const cusparseSolvePolicy_t policy_U = CUSPARSE_SOLVE_POLICY_USE_LEVEL;
const cusparseOperation_t trans_L  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseOperation_t trans_U  = CUSPARSE_OPERATION_NON_TRANSPOSE;
const cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;

// step 1: create a descriptor which contains
// - matrix M is base-1
// - matrix L is base-1
// - matrix L is lower triangular
// - matrix L has unit diagonal
// - matrix U is base-1
// - matrix U is upper triangular
// - matrix U has non-unit diagonal
cusparseCreateMatDescr(&descr_M);
cusparseSetMatIndexBase(descr_M, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_M, CUSPARSE_MATRIX_TYPE_GENERAL);

cusparseCreateMatDescr(&descr_L);
cusparseSetMatIndexBase(descr_L, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_L, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_L, CUSPARSE_FILL_MODE_LOWER);
cusparseSetMatDiagType(descr_L, CUSPARSE_DIAG_TYPE_UNIT);

cusparseCreateMatDescr(&descr_U);
cusparseSetMatIndexBase(descr_U, CUSPARSE_INDEX_BASE_ONE);
cusparseSetMatType(descr_U, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatFillMode(descr_U, CUSPARSE_FILL_MODE_UPPER);
cusparseSetMatDiagType(descr_U, CUSPARSE_DIAG_TYPE_NON_UNIT);

// step 2: create a empty info structure
// we need one info for bsrilu02 and two info's for bsrsv2
cusparseCreateBsrilu02Info(&info_M);
cusparseCreateBsrsv2Info(&info_L);
cusparseCreateBsrsv2Info(&info_U);

// step 3: query how much memory used in bsrilu02 and bsrsv2, and allocate the
 buffer
cusparseDbsrilu02_bufferSize(handle, dir, mb, nnzb,
    descr_M, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_M,
 &pBufferSize_M);
cusparseDbsrsv2_bufferSize(handle, dir, trans_L, mb, nnzb,
    descr_L, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_L,
 &pBufferSize_L);
cusparseDbsrsv2_bufferSize(handle, dir, trans_U, mb, nnzb,
    descr_U, d_bsrVal, d_bsrRowPtr, d_bsrColInd, blockDim, info_U,
```

The function supports the following properties if **`pBuffer != NULL`**

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`dirA`	storage format of blocks: either **`CUSPARSE_DIRECTION_ROW`** or **`CUSPARSE_DIRECTION_COLUMN`**.
`mb`	number of block rows and block columns of matrix **`A`**.
`nnzb`	number of nonzero blocks of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`**.
`bsrValA`	<type> array of **`nnzb`** $($= **`bsrRowPtrA(mb)`** $-$ **`bsrRowPtrA(0)`** $)$ nonzero blocks of matrix **`A`**.
`bsrRowPtrA`	integer array of **`mb`** $+1$ elements that contains the start of every block row and the end of the last block row plus one.
`bsrColIndA`	integer array of **`nnzb`** $($= **`bsrRowPtrA(mb)`** $-$ **`bsrRowPtrA(0)`** $)$ column indices of the nonzero blocks of matrix **`A`**.
`blockDim`	block dimension of sparse matrix A; must be larger than zero.
`info`	structure with information collected during the analysis phase (that should have been passed to the solve phase unchanged).
`policy`	the supported policies are **`CUSPARSE_SOLVE_POLICY_NO_LEVEL`** and **`CUSPARSE_SOLVE_POLICY_USE_LEVEL`**.
`pBuffer`	buffer allocated by the user; the size is returned by **`bsrilu02_bufferSize()`**.

Output

`bsrValA`	<type> matrix containing the incomplete-LU lower and upper triangular factors.

See `cusparseStatus_t` for the description of the return status

11.2.10. cusparseXbsrilu02_zeroPivot()

```
cusparseStatus_t
cusparseXbsrilu02_zeroPivot(cusparseHandle_t handle,
                            bsrilu02Info_t   info,
                            int*             position)
```

If the returned error code is **CUSPARSE_STATUS_ZERO_PIVOT**, **position=j** means **A(j,j)** has either a structural zero or a numerical zero (the block is not invertible). Otherwise **position=-1**.

The **position** can be 0-based or 1-based, the same as the matrix.

Function **cusparseXbsrilu02_zeroPivot()** is a blocking call. It calls **cudaDeviceSynchronize()** to make sure all previous kernels are done.

The **position** can be in the host memory or device memory. The user can set proper the mode with **cusparseSetPointerMode()**.

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
info	**info** contains structural zero or numerical zero if the user already called **bsrilu02_analysis()** or **bsrilu02()**.

Output

position	if no structural or numerical zero, **position** is -1; otherwise if **A(j,j)** is missing or **U(j,j)** is not invertible, **position=j**.

See `cusparseStatus_t` for the description of the return status

11.3. Tridiagonal Solve

Different algorithms for tridiagonal solve are discussed in this section.

11.3.1. cusparse<t>gtsv2_buffSizeExt()

```
cusparseStatus_t
cusparseSgtsv2_bufferSizeExt(cusparseHandle_t handle,
                             int              m,
                             int              n,
                             const float*     dl,
                             const float*     d,
                             const float*     du,
                             const float*     B,
                             int              ldb,
                             size_t*          bufferSizeInBytes)

cusparseStatus_t
cusparseDgtsv2_bufferSizeExt(cusparseHandle_t handle,
                             int              m,
                             int              n,
                             const double*    dl,
                             const double*    d,
                             const double*    du,
                             const double*    B,
                             int              ldb,
                             size_t*          bufferSizeInBytes)

cusparseStatus_t
cusparseCgtsv2_bufferSizeExt(cusparseHandle_t handle,
                             int              m,
                             int              n,
                             const cuComplex* dl,
                             const cuComplex* d,
                             const cuComplex* du,
                             const cuComplex* B,
                             int              ldb,
                             size_t*          bufferSizeInBytes)

cusparseStatus_t
cusparseZgtsv2_bufferSizeExt(cusparseHandle_t       handle,
                             int                    m,
                             int                    n,
                             const cuDoubleComplex* dl,
                             const cuDoubleComplex* d,
                             const cuDoubleComplex* du,
                             const cuDoubleComplex* B,
                             int                    ldb,
                             size_t*                bufferSizeInBytes)
```

This function returns the size of the buffer used in **`gtsv2`** which computes the solution of a tridiagonal linear system with multiple right-hand sides.

$$A * X = B$$

The coefficient matrix **`A`** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**`dl`**), main (**`d`**), and upper (**`du`**) matrix diagonals; the right-hand sides are stored in the dense matrix **`B`**. Notice that solution **`X`** overwrites right-hand-side matrix **`B`** on exit.

- The routine requires no extra storage

- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	the size of the linear system (must be ≥ 3).
`n`	number of right-hand sides, columns of matrix `B`.
`dl`	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
`d`	<type> dense array containing the main diagonal of the tri-diagonal linear system.
`du`	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
`B`	<type> dense right-hand-side array of dimensions `(ldb, n)`.
`ldb`	leading dimension of `B` (that is ≥ $\max(1, m)$).

Output

`pBufferSizeInBytes`	number of bytes of the buffer used in the `gtsv2`.

See `cusparseStatus_t` for the description of the return status

11.3.2. cusparse<t>gtsv2()

```
cusparseStatus_t
cusparseSgtsv2(cusparseHandle_t handle,
               int              m,
               int              n,
               const float*     dl,
               const float*     d,
               const float*     du,
               float*           B,
               int              ldb,
               void             pBuffer)

cusparseStatus_t
cusparseDgtsv2(cusparseHandle_t handle,
               int              m,
               int              n,
               const double*    dl,
               const double*    d,
               const double*    du,
               double*          B,
               int              ldb,
               void             pBuffer)

cusparseStatus_t
cusparseCgtsv2(cusparseHandle_t handle,
               int              m,
               int              n,
               const cuComplex* dl,
               const cuComplex* d,
               const cuComplex* du,
               cuComplex*       B,
               int              ldb,
               void             pBuffer)

cusparseStatus_t
cusparseZgtsv2(cusparseHandle_t       handle,
               int                    m,
               int                    n,
               const cuDoubleComplex* dl,
               const cuDoubleComplex* d,
               const cuDoubleComplex* du,
               cuDoubleComplex*       B,
               int                    ldb,
               void                   pBuffer)
```

This function computes the solution of a tridiagonal linear system with multiple right-hand sides:

$$A * X = B$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

Assuming **A** is of size **m** and base-1, **dl**, **d** and **du** are defined by the following formula:

dl(i) := A(i, i-1) for **i=1,2,...,m**

The first element of dl is out-of-bound (**dl(1) := A(1,0)**), so **dl(1) = 0**.

d(i) = A(i,i) for **i=1,2,...,m**

du(i) = A(i,i+1) for **i=1,2,...,m**

The last element of du is out-of-bound (**du(m) := A(m,m+1)**), so **du(m) = 0**.

The routine does perform pivoting, which usually results in more accurate and more stable results than **cusparse<t>gtsv_nopivot()** or **cusparse<t>gtsv2_nopivot()** at the expense of some execution time.

This function requires a buffer size returned by **gtsv2_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B** (that is ≥ $\max(1, m)$).
pBuffer	buffer allocated by the user, the size is return by **gtsv2_bufferSizeExt**.

Output

B	<type> dense solution array of dimensions **(ldb, n)**.

See `cusparseStatus_t` for the description of the return status

11.3.3. cusparse<t>gtsv2_nopivot_bufferSizeExt()

```
cusparseStatus_t
cusparseSgtsv2_nopivot_bufferSizeExt(cusparseHandle_t handle,
                                     int              m,
                                     int              n,
                                     const float*     dl,
                                     const float*     d,
                                     const float*     du,
                                     const float*     B,
                                     int              ldb,
                                     size_t*          bufferSizeInBytes)

cusparseStatus_t
cusparseDgtsv2_nopivot_bufferSizeExt(cusparseHandle_t handle,
                                     int              m,
                                     int              n,
                                     const double*    dl,
                                     const double*    d,
                                     const double*    du,
                                     const double*    B,
                                     int              ldb,
                                     size_t*          bufferSizeInBytes)

cusparseStatus_t
cusparseCgtsv2_nopivot_bufferSizeExt(cusparseHandle_t handle,
                                     int              m,
                                     int              n,
                                     const cuComplex* dl,
                                     const cuComplex* d,
                                     const cuComplex* du,
                                     const cuComplex* B,
                                     int              ldb,
                                     size_t*          bufferSizeInBytes)

cusparseStatus_t
cusparseZgtsv2_nopivot_bufferSizeExt(cusparseHandle_t       handle,
                                     int                    m,
                                     int                    n,
                                     const cuDoubleComplex* dl,
                                     const cuDoubleComplex* d,
                                     const cuDoubleComplex* du,
                                     const cuDoubleComplex* B,
                                     int                    ldb,
                                     size_t*
 bufferSizeInBytes)
```

This function returns the size of the buffer used in **`gtsv2_nopivot`** which computes the solution of a tridiagonal linear system with multiple right-hand sides.

$$A * X = B$$

The coefficient matrix **`A`** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**`dl`**), main (**`d`**), and upper (**`du`**) matrix diagonals; the right-hand sides are stored in the dense matrix **`B`**. Notice that solution **`X`** overwrites right-hand-side matrix **`B`** on exit.

- The routine requires no extra storage

- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B**. (that is ≥ $\max(1, m)$).

Output

pBufferSizeInBytes	number of bytes of the buffer used in the **gtsv2_nopivot**.

See `cusparseStatus_t` for the description of the return status

11.3.4. cusparse<t>gtsv2_nopivot()

```
cusparseStatus_t
cusparseSgtsv2_nopivot(cusparseHandle_t handle,
                       int              m,
                       int              n,
                       const float*     dl,
                       const float*     d,
                       const float*     du,
                       float*           B,
                       int              ldb,
                       void*            pBuffer)

cusparseStatus_t
cusparseDgtsv2_nopivot(cusparseHandle_t handle,
                       int              m,
                       int              n,
                       const double*    dl,
                       const double*    d,
                       const double*    du,
                       double*          B,
                       int              ldb,
                       void*            pBuffer)

cusparseStatus_t
cusparseCgtsv2_nopivot(cusparseHandle_t handle,
                       int              m,
                       int              n,
                       const cuComplex* dl,
                       const cuComplex* d,
                       const cuComplex* du,
                       cuComplex*       B,
                       int              ldb,
                       void*            pBuffer)

cusparseStatus_t
cusparseZgtsv2_nopivot(cusparseHandle_t       handle,
                       int                    m,
                       int                    n,
                       const cuDoubleComplex* dl,
                       const cuDoubleComplex* d,
                       const cuDoubleComplex* du,
                       cuDoubleComplex*       B,
                       int                    ldb,
                       void*                  pBuffer)
```

This function computes the solution of a tridiagonal linear system with multiple right-hand sides:

$$A * X = B$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

The routine does not perform any pivoting and uses a combination of the Cyclic Reduction (CR) and the Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves better performance when **m** is a power of 2.

This function requires a buffer size returned by **gtsv2_nopivot_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	the size of the linear system (must be ≥ 3).
n	number of right-hand sides, columns of matrix **B**.
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
B	<type> dense right-hand-side array of dimensions **(ldb, n)**.
ldb	leading dimension of **B**. (that is ≥ $\max(1, m)$).
pBuffer	buffer allocated by the user, the size is return by **gtsv2_nopivot_bufferSizeExt**.

Output

B	<type> dense solution array of dimensions **(ldb, n)**.

See `cusparseStatus_t` for the description of the return status

11.4. Batched Tridiagonal Solve

Different algorithms for batched tridiagonal solve are discussed in this section.

11.4.1. cusparse<t>gtsv2StridedBatch_bufferSizeExt()

```
cusparseStatus_t
cusparseSgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
                                         int              m,
                                         const float*     dl,
                                         const float*     d,
                                         const float*     du,
                                         const float*     x,
                                         int              batchCount,
                                         int              batchStride,
                                         size_t*
 bufferSizeInBytes)

cusparseStatus_t
cusparseDgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
                                         int              m,
                                         const double*    dl,
                                         const double*    d,
                                         const double*    du,
                                         const double*    x,
                                         int              batchCount,
                                         int              batchStride,
                                         size_t*
 bufferSizeInBytes)

cusparseStatus_t
cusparseCgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t handle,
                                         int              m,
                                         const cuComplex* dl,
                                         const cuComplex* d,
                                         const cuComplex* du,
                                         const cuComplex* x,
                                         int              batchCount,
                                         int              batchStride,
                                         size_t*
 bufferSizeInBytes)

cusparseStatus_t
cusparseZgtsv2StridedBatch_bufferSizeExt(cusparseHandle_t       handle,
                                         int                    m,
                                         const cuDoubleComplex* dl,
                                         const cuDoubleComplex* d,
                                         const cuDoubleComplex* du,
                                         const cuDoubleComplex* x,
                                         int
 batchCount,
                                         int
 batchStride,
                                         size_t*
 bufferSizeInBytes)
```

This function returns the size of the buffer used in **`gtsv2StridedBatch`** which computes the solution of multiple tridiagonal linear systems for i=0,…,**`batchCount`**:

$$A^{(i)} * y^{(i)} = x^{(i)}$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **X**. Notice that solution **Y** overwrites right-hand-side matrix **X** on exit. The different matrices are assumed to be of the same size and are stored with a fixed **batchStride** in memory.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
n	the size of the linear system (must be ≥ 3).
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal $dl^{(i)}$ that corresponds to the i^{th} linear system starts at location **dl+batchStride×i** in memory. Also, the first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system. The main diagonal $d^{(i)}$ that corresponds to the i^{th} linear system starts at location **d+batchStride×i** in memory.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The upper diagonal $du^{(i)}$ that corresponds to the i^{th} linear system starts at location **du+batchStride×i** in memory. Also, the last element of each upper diagonal must be zero.
x	<type> dense array that contains the right-hand-side of the tri-diagonal linear system. The right-hand-side $x^{(i)}$ that corresponds to the i^{th} linear system starts at location **x+batchStride×i**in memory.
batchCount	number of systems to solve.
batchStride	stride (number of elements) that separates the vectors of every system (must be at least **m**).

Output

pBufferSizeInBytes	number of bytes of the buffer used in the **gtsv2StridedBatch**.

See `cusparseStatus_t` for the description of the return status

11.4.2. cusparse<t>gtsv2StridedBatch()

```
cusparseStatus_t
cusparseSgtsv2StridedBatch(cusparseHandle_t handle,
                           int              m,
                           const float*     dl,
                           const float*     d,
                           const float*     du,
                           float*           x,
                           int              batchCount,
                           int              batchStride,
                           void*            pBuffer)

cusparseStatus_t
cusparseDgtsv2StridedBatch(cusparseHandle_t handle,
                           int              m,
                           const double*    dl,
                           const double*    d,
                           const double*    du,
                           double*          x,
                           int              batchCount,
                           int              batchStride,
                           void*            pBuffer)

cusparseStatus_t
cusparseCgtsv2StridedBatch(cusparseHandle_t handle,
                           int              m,
                           const cuComplex* dl,
                           const cuComplex* d,
                           const cuComplex* du,
                           cuComplex*       x,
                           int              batchCount,
                           int              batchStride,
                           void*            pBuffer)

cusparseStatus_t
cusparseZgtsv2StridedBatch(cusparseHandle_t       handle,
                           int                    m,
                           const cuDoubleComplex* dl,
                           const cuDoubleComplex* d,
                           const cuDoubleComplex* du,
                           cuDoubleComplex*       x,
                           int                    batchCount,
                           int                    batchStride,
                           void*                  pBuffer)
```

This function computes the solution of multiple tridiagonal linear systems for i=0, …,**batchCount**:

$$A^{(i)} * \mathrm{y}^{(i)} = \mathrm{x}^{(i)}$$

The coefficient matrix **A** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**dl**), main (**d**), and upper (**du**) matrix diagonals; the right-hand sides are stored in the dense matrix **X**. Notice that solution **Y** overwrites right-hand-side matrix **X** on exit. The different matrices are assumed to be of the same size and are stored with a fixed **batchStride** in memory.

The routine does not perform any pivoting and uses a combination of the Cyclic Reduction (CR) and the Parallel Cyclic Reduction (PCR) algorithms to find the solution. It achieves better performance when **m** is a power of 2.

This function requires a buffer size returned by **gtsv2StridedBatch_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
n	the size of the linear system (must be ≥ 3).
dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The lower diagonal $dl^{(i)}$ that corresponds to the i^{th} linear system starts at location **dl+batchStride×i** in memory. Also, the first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system. The main diagonal $d^{(i)}$ that corresponds to the i^{th} linear system starts at location **d+batchStride×i** in memory.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The upper diagonal $du^{(i)}$ that corresponds to the i^{th} linear system starts at location **du+batchStride×i** in memory. Also, the last element of each upper diagonal must be zero.
x	<type> dense array that contains the right-hand-side of the tri-diagonal linear system. The right-hand-side $x^{(i)}$ that corresponds to the i^{th} linear system starts at location **x+batchStride×i**in memory.
batchCount	number of systems to solve.
batchStride	stride (number of elements) that separates the vectors of every system (must be at least **n**).
pBuffer	buffer allocated by the user, the size is return by **gtsv2StridedBatch_bufferSizeExt**.

Output

x	<type> dense array that contains the solution of the tri-diagonal linear system. The solution $x^{(i)}$ that corresponds to the i^{th} linear system starts at location **x+batchStride×i**in memory.

See `cusparseStatus_t` for the description of the return status

11.4.3. cusparse<t>gtsvInterleavedBatch()

```
cusparseStatus_t
cusparseSgtsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
                                             int              algo,
                                             int              m,
                                             const float*     dl,
                                             const float*     d,
                                             const float*     du,
                                             const float*     x,
                                             int              batchCount,
                                             size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseDgtsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
                                             int              algo,
                                             int              m,
                                             const double*    dl,
                                             const double*    d,
                                             const double*    du,
                                             const double*    x,
                                             int              batchCount,
                                             size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseCgtsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
                                             int              algo,
                                             int              m,
                                             const cuComplex* dl,
                                             const cuComplex* d,
                                             const cuComplex* du,
                                             const cuComplex* x,
                                             int              batchCount,
                                             size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseZgtsvInterleavedBatch_bufferSizeExt(cusparseHandle_t
 handle,
                                             int                    algo,
                                             int                    m,
                                             const cuDoubleComplex* dl,
                                             const cuDoubleComplex* d,
                                             const cuDoubleComplex* du,
                                             const cuDoubleComplex* x,
                                             int
 batchCount,
                                             size_t*
 pBufferSizeInBytes)
```

```
cusparseStatus_t
cusparseSgtsvInterleavedBatch(cusparseHandle_t handle,
                              int              algo,
                              int              m,
                              float*           dl,
                              float*           d,
                              float*           du,
                              float*           x,
                              int              batchCount,
                              void*            pBuffer)

cusparseStatus_t
cusparseDgtsvInterleavedBatch(cusparseHandle_t handle,
```

This function computes the solution of multiple tridiagonal linear systems for i=0, …,**`batchCount`**:

$$A^{(i)} * \mathrm{x}^{(i)} = \mathrm{b}^{(i)}$$

The coefficient matrix **`A`** of each of these tri-diagonal linear system is defined with three vectors corresponding to its lower (**`dl`**), main (**`d`**), and upper (**`du`**) matrix diagonals; the right-hand sides are stored in the dense matrix **`B`**. Notice that solution **`X`** overwrites right-hand-side matrix **`B`** on exit.

Assuming **`A`** is of size **`m`** and base-1, **`dl`**, **`d`** and **`du`** are defined by the following formula:

`dl(i) := A(i, i-1)` for **`i=1,2,...,m`**

The first element of dl is out-of-bound (**`dl(1) := A(1,0)`**), so **`dl(1) = 0`**.

`d(i) = A(i,i)` for **`i=1,2,...,m`**

`du(i) = A(i,i+1)` for **`i=1,2,...,m`**

The last element of du is out-of-bound (**`du(m) := A(m,m+1)`**), so **`du(m) = 0`**.

The data layout is different from **`gtsvStridedBatch`** which aggregates all matrices one after another. Instead, **`gtsvInterleavedBatch`** gathers different matrices of the same element in a continous manner. If **`dl`** is regarded as a 2-D array of size **`m-by-batchCount`**, **`dl(:,j)`** to store **`j-th`** matrix. **`gtsvStridedBatch`** uses column-major while **`gtsvInterleavedBatch`** uses row-major.

The routine provides three different algorithms, selected by parameter **`algo`**. The first algorithm is **`cuThomas`** provided by **`Barcelona Supercomputing Center`**. The second algorithm is LU with partial pivoting and last algorithm is QR. From stability perspective, cuThomas is not numerically stable because it does not have pivoting. LU with partial pivoting and QR are stable. From performance perspective, LU with partial pivoting and QR is about 10% to 20% slower than cuThomas.

This function requires a buffer size returned by **`gtsvInterleavedBatch_bufferSizeExt()`**. The address of **`pBuffer`** must be multiple of 128 bytes. If it is not, **`CUSPARSE_STATUS_INVALID_VALUE`** is returned.

If the user prepares aggregate format, one can use **`cublasXgeam`** to get interleaved format. However such transformation takes time comparable to solver itself. To reach best performance, the user must prepare interleaved format explicitly.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`algo`	algo = 0: cuThomas (unstable algorithm); algo = 1: LU with pivoting (stable algorithm); algo = 2: QR (stable algorithm)
`m`	the size of the linear system.

dl	<type> dense array containing the lower diagonal of the tri-diagonal linear system. The first element of each lower diagonal must be zero.
d	<type> dense array containing the main diagonal of the tri-diagonal linear system.
du	<type> dense array containing the upper diagonal of the tri-diagonal linear system. The last element of each upper diagonal must be zero.
x	<type> dense right-hand-side array of dimensions **(batchCount, n)**.
pBuffer	buffer allocated by the user, the size is return by **gtsvInterleavedBatch_bufferSizeExt**.

Output

x	<type> dense solution array of dimensions **(batchCount, n)**.

See `cusparseStatus_t` for the description of the return status

11.5. Batched Pentadiagonal Solve

Different algorithms for batched pentadiagonal solve are discussed in this section.

11.5.1. cusparse<t>gpsvInterleavedBatch()

```
cusparseStatus_t
cusparseSgpsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
                                            int              algo,
                                            int              m,
                                            const float*     ds,
                                            const float*     dl,
                                            const float*     d,
                                            const float*     du,
                                            const float*     dw,
                                            const float*     x,
                                            int              batchCount,
                                            size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseDgpsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
                                            int              algo,
                                            int              m,
                                            const double*    ds,
                                            const double*    dl,
                                            const double*    d,
                                            const double*    du,
                                            const double*    dw,
                                            const double*    x,
                                            int              batchCount,
                                            size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseCgpsvInterleavedBatch_bufferSizeExt(cusparseHandle_t handle,
                                            int              algo,
                                            int              m,
                                            const cuComplex* ds,
                                            const cuComplex* dl,
                                            const cuComplex* d,
                                            const cuComplex* du,
                                            const cuComplex* dw,
                                            const cuComplex* x,
                                            int              batchCount,
                                            size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseZgpsvInterleavedBatch_bufferSizeExt(cusparseHandle_t
 handle,
                                            int                    algo,
                                            int                    m,
                                            const cuDoubleComplex* ds,
                                            const cuDoubleComplex* dl,
                                            const cuDoubleComplex* d,
                                            const cuDoubleComplex* du,
                                            const cuDoubleComplex* dw,
                                            const cuDoubleComplex* x,
                                            int
 batchCount,
                                            size_t*
 pBufferSizeInBytes)
```

```
cusparseStatus_t
cusparseSgpsvInterleavedBatch(cusparseHandle_t handle,
                              int              algo,
                              int              m,
                              float*           ds,
                              float*           dl,
```

This function computes the solution of multiple penta-diagonal linear systems for i=0, …,**batchCount**:

$$A^{(i)} * x^{(i)} = b^{(i)}$$

The coefficient matrix **A** of each of these penta-diagonal linear system is defined with five vectors corresponding to its lower (**ds, dl**), main (**d**), and upper (**du, dw**) matrix diagonals; the right-hand sides are stored in the dense matrix **B**. Notice that solution **X** overwrites right-hand-side matrix **B** on exit.

Assuming **A** is of size **m** and base-1, **ds**, **dl**, **d**, **du** and **dw** are defined by the following formula:

ds(i) := A(i, i-2) for **i=1,2,...,m**

The first two elements of ds is out-of-bound (**ds(1) := A(1,-1)**, **ds(2) := A(2,0)**), so **ds(1) = 0** and **ds(2) = 0**.

dl(i) := A(i, i-1) for **i=1,2,...,m**

The first element of dl is out-of-bound (**dl(1) := A(1,0)**), so **dl(1) = 0**.

d(i) = A(i,i) for **i=1,2,...,m**

du(i) = A(i,i+1) for **i=1,2,...,m**

The last element of du is out-of-bound (**du(m) := A(m,m+1)**), so **du(m) = 0**.

dw(i) = A(i,i+2) for **i=1,2,...,m**

The last two elements of dw is out-of-bound (**dw(m-1) := A(m-1,m+1)**, **dw(m) := A(m,m+2)**), so **dw(m-1) = 0** and **dw(m) = 0**.

The data layout is the same as **gtsvStridedBatch**.

The routine is numerically stable because it uses QR to solve the linear system.

This function requires a buffer size returned by **gpsvInterleavedBatch_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If it is not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Appendix section shows an example of **gpsvInterleavedBatch**. If the user prepares aggregate format, one can use **cublasXgeam** to get interleaved format. However such transformation takes time comparable to solver itself. To reach best performance, the user must prepare interleaved format explicitly.

The function supports the following properties if **pBuffer != NULL**

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
algo	only support algo = 0 (QR)
m	the size of the linear system.

ds	<type> dense array containing the lower diagonal (distance 2 to the diagonal) of the penta-diagonal linear system. The first two elements must be zero.
dl	<type> dense array containing the lower diagonal (distance 1 to the diagonal) of the penta-diagonal linear system. The first element must be zero.
d	<type> dense array containing the main diagonal of the penta-diagonal linear system.
du	<type> dense array containing the upper diagonal (distance 1 to the diagonal) of the penta-diagonal linear system. The last element must be zero.
dw	<type> dense array containing the upper diagonal (distance 2 to the diagonal) of the penta-diagonal linear system. The last two elements must be zero.
x	<type> dense right-hand-side array of dimensions **(batchCount, n).**
pBuffer	buffer allocated by the user, the size is return by **gpsvInterleavedBatch_bufferSizeExt**.

Output

x	<type> dense solution array of dimensions **(batchCount, n).**

See `cusparseStatus_t` for the description of the return status

Chapter 12. CUSPARSE REORDERINGS REFERENCE

This chapter describes the reordering routines used to manipulate sparse matrices.

12.1. cusparse<t>csrcolor()

```
cusparseStatus_t
cusparseScsrcolor(cusparseHandle_t         handle,
                  int                      m,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  const float*             csrValA,
                  const int*               csrRowPtrA,
                  const int*               csrColIndA,
                  const float*             fractionToColor,
                  int*                     ncolors,
                  int*                     coloring,
                  int*                     reordering,
                  cusparseColorInfo_t      info)

cusparseStatus_t
cusparseDcsrcolor(cusparseHandle_t         handle,
                  int                      m,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  const double*            csrValA,
                  const int*               csrRowPtrA,
                  const int*               csrColIndA,
                  const double*            fractionToColor,
                  int*                     ncolors,
                  int*                     coloring,
                  int*                     reordering,
                  cusparseColorInfo_t      info)

cusparseStatus_t
cusparseCcsrcolor(cusparseHandle_t         handle,
                  int                      m,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  const cuComplex*         csrValA,
                  const int*               csrRowPtrA,
                  const int*               csrColIndA,
                  const cuComplex*         fractionToColor,
                  int*                     ncolors,
                  int*                     coloring,
                  int*                     reordering,
                  cusparseColorInfo_t      info)

cusparseStatus_t
cusparseZcsrcolor(cusparseHandle_t         handle,
                  int                      m,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  const cuDoubleComplex*   csrValA,
                  const int*               csrRowPtrA,
                  const int*               csrColIndA,
                  const cuDoubleComplex*   fractionToColor,
                  int*                     ncolors,
                  int*                     coloring,
                  int*                     reordering,
                  cusparseColorInfo_t      info)
```

This function performs the coloring of the adjacency graph associated with the matrix A stored in CSR format. The coloring is an assignment of colors (integer numbers) to nodes, such that neighboring nodes have distinct colors. An approximate coloring algorithm is used in this routine, and is stopped when a certain percentage of nodes has been colored. The rest of the nodes are assigned distinct colors (an increasing sequence of integers numbers, starting from the last integer used previously). The last two auxiliary routines can be used to extract the resulting number of colors, their assignment and the associated reordering. The reordering is such that nodes that have been assigned the same color are reordered to be next to each other.

The matrix A passed to this routine, must be stored as a general matrix and have a symmetric sparsity pattern. If the matrix is nonsymmetric the user should pass A+A^T as a parameter to this routine.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows of matrix `A`.
`nnz`	number of nonzero elements of matrix `A`.
`descrA`	the descriptor of matrix `A`. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL`. Also, the supported index bases are `CUSPARSE_INDEX_BASE_ZERO` and `CUSPARSE_INDEX_BASE_ONE`.
`csrValA`	<type> array of `nnz` (= `csrRowPtrA(m)` - `csrRowPtrA(0)`) nonzero elements of matrix `A`.
`csrRowPtrA`	integer array of `m+1` elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of `nnz` (= `csrRowPtrA(m)` - `csrRowPtrA(0)`) column indices of the nonzero elements of matrix `A`.
`fractionToColor`	fraction of nodes to be colored, which should be in the interval [0.0,1.0], for example 0.8 implies that 80 percent of nodes will be colored.
`info`	structure with information to be passed to the coloring.

Output

`ncolors`	The number of distinct colors used (at most the size of the matrix, but likely much smaller).
`coloring`	The resulting coloring permutation
`reordering`	The resulting reordering permutation (untouched if NULL)

See `cusparseStatus_t` for the description of the return status

Chapter 13. CUSPARSE FORMAT CONVERSION REFERENCE

This chapter describes the conversion routines between different sparse and dense storage formats.

coosort, **csrsort**, **cscsort**, and **csru2csr** are sorting routines without malloc inside, the following table estimates the buffer size

routine	buffer size	maximum problem size if buffer is limited by 2GB
coosort	> 16*n bytes	125M
csrsort or cscsort	> 20*n bytes	100M
csru2csr	'd' > 28*n bytes ; 'z' > 36*n bytes	71M for 'd' and 55M for 'z'

13.1. cusparse<t>bsr2csr()

```
cusparseStatus_t
cusparseSbsr2csr(cusparseHandle_t         handle,
                 cusparseDirection_t      dir,
                 int                      mb,
                 int                      nb,
                 const cusparseMatDescr_t descrA,
                 const float*             bsrValA,
                 const int*               bsrRowPtrA,
                 const int*               bsrColIndA,
                 int                      blockDim,
                 const cusparseMatDescr_t descrC,
                 float*                   csrValC,
                 int*                     csrRowPtrC,
                 int*                     csrColIndC)

cusparseStatus_t
cusparseDbsr2csr(cusparseHandle_t         handle,
                 cusparseDirection_t      dir,
                 int                      mb,
                 int                      nb,
                 const cusparseMatDescr_t descrA,
                 const double*            bsrValA,
                 const int*               bsrRowPtrA,
                 const int*               bsrColIndA,
                 int                      blockDim,
                 const cusparseMatDescr_t descrC,
                 double*                  csrValC,
                 int*                     csrRowPtrC,
                 int*                     csrColIndC)

cusparseStatus_t
cusparseCbsr2csr(cusparseHandle_t         handle,
                 cusparseDirection_t      dir,
                 int                      mb,
                 int                      nb,
                 const cusparseMatDescr_t descrA,
                 const cuComplex*         bsrValA,
                 const int*               bsrRowPtrA,
                 const int*               bsrColIndA,
                 int                      blockDim,
                 const cusparseMatDescr_t descrC,
                 cuComplex*               csrValC,
                 int*                     csrRowPtrC,
                 int*                     csrColIndC)

cusparseStatus_t
cusparseZbsr2csr(cusparseHandle_t         handle,
                 cusparseDirection_t      dir,
                 int                      mb,
                 int                      nb,
                 const cusparseMatDescr_t descrA,
                 const cuDoubleComplex*   bsrValA,
                 const int*               bsrRowPtrA,
                 const int*               bsrColIndA,
                 int                      blockDim,
                 const cusparseMatDescr_t descrC,
                 cuDoubleComplex*         csrValC,
                 int*                     csrRowPtrC,
                 int*                     csrColIndC)
```

This function converts a sparse matrix in BSR format that is defined by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA**) into a sparse matrix in CSR format that is defined by arrays **csrValC**, **csrRowPtrC**, and **csrColIndC**.

Let **m(=mb*blockDim)** be the number of rows of **A** and **n(=nb*blockDim)** be number of columns of **A**, then **A** and **C** are **m*n** sparse matrices. The BSR format of **A** contains **nnzb(=bsrRowPtrA[mb] - bsrRowPtrA[0])** nonzero blocks, whereas the sparse matrix **A** contains **nnz(=nnzb*blockDim*blockDim)** elements. The user must allocate enough space for arrays **csrRowPtrC**, **csrColIndC**, and **csrValC**. The requirements are as follows:

csrRowPtrC of **m+1** elements

csrValC of **nnz** elements

csrColIndC of **nnz** elements

The general procedure is as follows:

```
// Given BSR format (bsrRowPtrA, bsrcolIndA, bsrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int m = mb*blockDim;
int nnzb = bsrRowPtrA[mb] - bsrRowPtrA[0]; // number of blocks
int nnz  = nnzb * blockDim * blockDim; // number of elements
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnz);
cudaMalloc((void**)&csrValC, sizeof(float)*nnz);
cusparseSbsr2csr(handle, dir, mb, nb,
        descrA,
        bsrValA, bsrRowPtrA, bsrColIndA,
        blockDim,
        descrC,
        csrValC, csrRowPtrC, csrColIndC);
```

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution if **blockDim == 1**
- The routine does ***not*** support CUDA graph capture if **blockDim == 1**

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows of sparse matrix **A**.
nb	number of block columns of sparse matrix **A**.
descrA	the descriptor of matrix **A**.
bsrValA	<type> array of **nnzb*blockDim*blockDim** nonzero elements of matrix **A**.
bsrRowPtrA	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **A**.
bsrColIndA	integer array of **nnzb** column indices of the nonzero blocks of matrix **A**.

blockDim	block dimension of sparse matrix **A**.
descrC	the descriptor of matrix **C**.

Output

csrValC	<type> array of **nnz(=csrRowPtrC[m]-csrRowPtrC[0])** nonzero elements of matrix **C**.
csrRowPtrC	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one of matrix **C**.
csrColIndC	integer array of **nnz** column indices of the nonzero elements of matrix **C**.

See `cusparseStatus_t` for the description of the return status

13.2. cusparse<t>gebsr2gebsc()

```
cusparseStatus_t
cusparseSgebsr2gebsc_bufferSize(cusparseHandle_t handle,
                                int              mb,
                                int              nb,
                                int              nnzb,
                                const float*     bsrVal,
                                const int*       bsrRowPtr,
                                const int*       bsrColInd,
                                int              rowBlockDim,
                                int              colBlockDim,
                                int*             pBufferSize)

cusparseStatus_t
cusparseDgebsr2gebsc_bufferSize(cusparseHandle_t handle,
                                int              mb,
                                int              nb,
                                int              nnzb,
                                const double*    bsrVal,
                                const int*       bsrRowPtr,
                                const int*       bsrColInd,
                                int              rowBlockDim,
                                int              colBlockDim,
                                int*             pBufferSize)

cusparseStatus_t
cusparseCgebsr2gebsc_bufferSize(cusparseHandle_t handle,
                                int              mb,
                                int              nb,
                                int              nnzb,
                                const cuComplex* bsrVal,
                                const int*       bsrRowPtr,
                                const int*       bsrColInd,
                                int              rowBlockDim,
                                int              colBlockDim,
                                int*             pBufferSize)

cusparseStatus_t
cusparseZgebsr2gebsc_bufferSize(cusparseHandle_t      handle,
                                int                   mb,
                                int                   nb,
                                int                   nnzb,
                                const cuDoubleComplex* bsrVal,
                                const int*            bsrRowPtr,
                                const int*            bsrColInd,
                                int                   rowBlockDim,
                                int                   colBlockDim,
                                int*                  pBufferSize)
```

```
cusparseStatus_t
cusparseSgebsr2gebsc(cusparseHandle_t    handle,
                     int                 mb,
                     int                 nb,
                     int                 nnzb,
                     const float*        bsrVal,
                     const int*          bsrRowPtr,
                     const int*          bsrColInd,
                     int                 rowBlockDim,
                     int                 colBlockDim,
                     float*              bscVal,
                     int*                bscRowInd,
                     int*                bscColPtr,
                     cusparseAction_t    copyValues,
```

This function can be seen as the same as **`csr2csc()`** when each block of size **`rowBlockDim*colBlockDim`** is regarded as a scalar.

This sparsity pattern of the result matrix can also be seen as the transpose of the original sparse matrix, but the memory layout of a block does not change.

The user must call **`gebsr2gebsc_bufferSize()`** to determine the size of the buffer required by **`gebsr2gebsc()`**, allocate the buffer, and pass the buffer pointer to **`gebsr2gebsc()`**.

- The routine requires no extra storage if **`pBuffer != NULL`**
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`mb`	number of block rows of sparse matrix **`A`**.
`nb`	number of block columns of sparse matrix **`A`**.
`nnzb`	number of nonzero blocks of matrix **`A`**.
`bsrVal`	<type> array of **`nnzb*rowBlockDim*colBlockDim`** nonzero elements of matrix **`A`**.
`bsrRowPtr`	integer array of **`mb+1`** elements that contains the start of every block row and the end of the last block row plus one.
`bsrColInd`	integer array of **`nnzb`** column indices of the non-zero blocks of matrix **`A`**.
`rowBlockDim`	number of rows within a block of **`A`**.
`colBlockDim`	number of columns within a block of **`A`**.
`copyValues`	**`CUSPARSE_ACTION_SYMBOLIC`** or **`CUSPARSE_ACTION_NUMERIC`**.
`baseIdx`	**`CUSPARSE_INDEX_BASE_ZERO`** or **`CUSPARSE_INDEX_BASE_ONE`**.
`pBufferSize`	host pointer containing number of bytes of the buffer used in **`gebsr2gebsc()`**.
`pBuffer`	buffer allocated by the user; the size is return by **`gebsr2gebsc_bufferSize()`**.

Output

`bscVal`	<type> array of **`nnzb*rowBlockDim*colBlockDim`** non-zero elements of matrix **`A`**. It is only filled-in if **`copyValues`** is set to **`CUSPARSE_ACTION_NUMERIC`**.
`bscRowInd`	integer array of **`nnzb`** row indices of the non-zero blocks of matrix **`A`**.

`bscColPtr`	integer array of `nb+1` elements that contains the start of every block column and the end of the last block column plus one.

See `cusparseStatus_t` for the description of the return status

13.3. cusparse<t>gebsr2gebsr()

```
cusparseStatus_t
cusparseSgebsr2gebsr_bufferSize(cusparseHandle_t         handle,
                                cusparseDirection_t      dir,
                                int                      mb,
                                int                      nb,
                                int                      nnzb,
                                const cusparseMatDescr_t descrA,
                                const float*             bsrValA,
                                const int*               bsrRowPtrA,
                                const int*               bsrColIndA,
                                int                      rowBlockDimA,
                                int                      colBlockDimA,
                                int                      rowBlockDimC,
                                int                      colBlockDimC,
                                int*                     pBufferSize)

cusparseStatus_t
cusparseDgebsr2gebsr_bufferSize(cusparseHandle_t         handle,
                                cusparseDirection_t      dir,
                                int                      mb,
                                int                      nb,
                                int                      nnzb,
                                const cusparseMatDescr_t descrA,
                                const double*            bsrValA,
                                const int*               bsrRowPtrA,
                                const int*               bsrColIndA,
                                int                      rowBlockDimA,
                                int                      colBlockDimA,
                                int                      rowBlockDimC,
                                int                      colBlockDimC,
                                int*                     pBufferSize)

cusparseStatus_t
cusparseCgebsr2gebsr_bufferSize(cusparseHandle_t         handle,
                                cusparseDirection_t      dir,
                                int                      mb,
                                int                      nb,
                                int                      nnzb,
                                const cusparseMatDescr_t descrA,
                                const cuComplex*         bsrValA,
                                const int*               bsrRowPtrA,
                                const int*               bsrColIndA,
                                int                      rowBlockDimA,
                                int                      colBlockDimA,
                                int                      rowBlockDimC,
                                int                      colBlockDimC,
                                int*                     pBufferSize)

cusparseStatus_t
cusparseZgebsr2gebsr_bufferSize(cusparseHandle_t         handle,
                                cusparseDirection_t      dir,
                                int                      mb,
                                int                      nb,
                                int                      nnzb,
                                const cusparseMatDescr_t descrA,
                                const cuDoubleComplex*   bsrValA,
                                const int*               bsrRowPtrA,
                                const int*               bsrColIndA,
                                int                      rowBlockDimA,
                                int                      colBlockDimA,
                                int                      rowBlockDimC,
                                int                      colBlockDimC,
                                int*                     pBufferSize)
```

This function converts a sparse matrix in general BSR format that is defined by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA** into a sparse matrix in another general BSR format that is defined by arrays **bsrValC**, **bsrRowPtrC**, and **bsrColIndC**.

If **rowBlockDimA=1** and **colBlockDimA=1**, **cusparse[S|D|C|Z]gebsr2gebsr()** is the same as **cusparse[S|D|C|Z]csr2gebsr()**.

If **rowBlockDimC=1** and **colBlockDimC=1**, **cusparse[S|D|C|Z]gebsr2gebsr()** is the same as **cusparse[S|D|C|Z]gebsr2csr()**.

A is an **m*n** sparse matrix where **m(=mb*rowBlockDim)** is the number of rows of **A**, and **n(=nb*colBlockDim)** is the number of columns of **A**. The general BSR format of **A** contains **nnzb(=bsrRowPtrA[mb] - bsrRowPtrA[0])** nonzero blocks. The matrix **C** is also general BSR format with a different block size, **rowBlockDimC*colBlockDimC**. If **m** is not a multiple of **rowBlockDimC**, or **n** is not a multiple of **colBlockDimC**, zeros are filled in. The number of block rows of **C** is **mc(=(m+rowBlockDimC-1)/rowBlockDimC)**. The number of block rows of **C** is **nc(=(n+colBlockDimC-1)/colBlockDimC)**. The number of nonzero blocks of **C** is **nnzc**.

The implementation adopts a two-step approach to do the conversion. First, the user allocates **bsrRowPtrC** of **mc+1** elements and uses function **cusparseXgebsr2gebsrNnz()** to determine the number of nonzero block columns per block row of matrix **C**. Second, the user gathers **nnzc** (number of non-zero block columns of matrix **C**) from either **(nnzc=*nnzTotalDevHostPtr)** or **(nnzc=bsrRowPtrC[mc]-bsrRowPtrC[0])** and allocates **bsrValC** of **nnzc*rowBlockDimC*colBlockDimC** elements and **bsrColIndC** of **nnzc** integers. Finally the function **cusparse[S|D|C|Z]gebsr2gebsr()** is called to complete the conversion.

The user must call **gebsr2gebsr_bufferSize()** to know the size of the buffer required by **gebsr2gebsr()**, allocate the buffer, and pass the buffer pointer to **gebsr2gebsr()**.

The general procedure is as follows:

```
// Given general BSR format (bsrRowPtrA, bsrColIndA, bsrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzc;
int m = mb*rowBlockDimA;
int n = nb*colBlockDimA;
int mc = (m+rowBlockDimC-1)/rowBlockDimC;
int nc = (n+colBlockDimC-1)/colBlockDimC;
int bufferSize;
void *pBuffer;
cusparseSgebsr2gebsr_bufferSize(handle, dir, mb, nb, nnzb,
    descrA, bsrValA, bsrRowPtrA, bsrColIndA,
    rowBlockDimA, colBlockDimA,
    rowBlockDimC, colBlockDimC,
    &bufferSize);
cudaMalloc((void**)&pBuffer, bufferSize);
cudaMalloc((void**)&bsrRowPtrC, sizeof(int)*(mc+1));
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzc;
cusparseXgebsr2gebsrNnz(handle, dir, mb, nb, nnzb,
    descrA, bsrRowPtrA, bsrColIndA,
    rowBlockDimA, colBlockDimA,
    descrC, bsrRowPtrC,
    rowBlockDimC, colBlockDimC,
    nnzTotalDevHostPtr,
    pBuffer);
if (NULL != nnzTotalDevHostPtr){
    nnzc = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzc, bsrRowPtrC+mc, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzc -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzc);
cudaMalloc((void**)&bsrValC, sizeof(float)*(rowBlockDimC*colBlockDimC)*nnzc);
cusparseSgebsr2gebsr(handle, dir, mb, nb, nnzb,
    descrA, bsrValA, bsrRowPtrA, bsrColIndA,
    rowBlockDimA, colBlockDimA,
    descrC, bsrValC, bsrRowPtrC, bsrColIndC,
    rowBlockDimC, colBlockDimC,
    pBuffer);
```

- The routines require no extra storage if **`pBuffer != NULL`**
- The routines do ***not*** support asynchronous execution
- The routines do ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`dir`	storage format of blocks, either **`CUSPARSE_DIRECTION_ROW`** or **`CUSPARSE_DIRECTION_COLUMN`**.
`mb`	number of block rows of sparse matrix **`A`**.
`nb`	number of block columns of sparse matrix **`A`**.
`nnzb`	number of nonzero blocks of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are

	CUSPARSE_INDEX_BASE_ZERO and CUSPARSE_INDEX_BASE_ONE.
bsrValA	<type> array of **nnzb*rowBlockDimA*colBlockDimA** non-zero elements of matrix **A**.
bsrRowPtrA	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **A**.
bsrColIndA	integer array of **nnzb** column indices of the non-zero blocks of matrix **A**.
rowBlockDimA	number of rows within a block of **A**.
colBlockDimA	number of columns within a block of **A**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
rowBlockDimC	number of rows within a block of **C**.
colBlockDimC	number of columns within a block of **C**.
pBufferSize	host pointer containing number of bytes of the buffer used in **gebsr2gebsr()**.
pBuffer	buffer allocated by the user; the size is return by **gebsr2gebsr_bufferSize()**.

Output

bsrValC	<type> array of **nnzc*rowBlockDimC*colBlockDimC** non-zero elements of matrix **C**.
bsrRowPtrC	integer array of **mc+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **C**.
bsrColIndC	integer array of **nnzc** block column indices of the nonzero blocks of matrix **C**.
nnzTotalDevHostPtr	total number of nonzero blocks of **C**. ***nnzTotalDevHostPtr** is the same as **bsrRowPtrC[mc]-bsrRowPtrC[0]**.

See `cusparseStatus_t` for the description of the return status

13.4. cusparse<t>gebsr2csr()

```
cusparseStatus_t
cusparseSgebsr2csr(cusparseHandle_t         handle,
                   cusparseDirection_t      dir,
                   int                      mb,
                   int                      nb,
                   const cusparseMatDescr_t descrA,
                   const float*             bsrValA,
                   const int*               bsrRowPtrA,
                   const int*               bsrColIndA,
                   int                      rowBlockDim,
                   int                      colBlockDim,
                   const cusparseMatDescr_t descrC,
                   float*                   csrValC,
                   int*                     csrRowPtrC,
                   int*                     csrColIndC)

cusparseStatus_t
cusparseDgebsr2csr(cusparseHandle_t         handle,
                   cusparseDirection_t      dir,
                   int                      mb,
                   int                      nb,
                   const cusparseMatDescr_t descrA,
                   const double*            bsrValA,
                   const int*               bsrRowPtrA,
                   const int*               bsrColIndA,
                   int                      rowBlockDim,
                   int                      colBlockDim,
                   const cusparseMatDescr_t descrC,
                   double*                  csrValC,
                   int*                     csrRowPtrC,
                   int*                     csrColIndC)

cusparseStatus_t
cusparseCgebsr2csr(cusparseHandle_t         handle,
                   cusparseDirection_t      dir,
                   int                      mb,
                   int                      nb,
                   const cusparseMatDescr_t descrA,
                   const cuComplex*         bsrValA,
                   const int*               bsrRowPtrA,
                   const int*               bsrColIndA,
                   int                      rowBlockDim,
                   int                      colBlockDim,
                   const cusparseMatDescr_t descrC,
                   cuComplex*               csrValC,
                   int*                     csrRowPtrC,
                   int*                     csrColIndC)

cusparseStatus_t
cusparseZgebsr2csr(cusparseHandle_t         handle,
                   cusparseDirection_t      dir,
                   int                      mb,
                   int                      nb,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex*   bsrValA,
                   const int*               bsrRowPtrA,
                   const int*               bsrColIndA,
                   int                      rowBlockDim,
                   int                      colBlockDim,
                   const cusparseMatDescr_t descrC,
                   cuDoubleComplex*         csrValC,
                   int*                     csrRowPtrC,
                   int*                     csrColIndC)
```

This function converts a sparse matrix in general BSR format that is defined by the three arrays **bsrValA**, **bsrRowPtrA**, and **bsrColIndA** into a sparse matrix in CSR format that is defined by arrays **csrValC**, **csrRowPtrC**, and **csrColIndC**.

Let **m(=mb*rowBlockDim)** be number of rows of **A** and **n(=nb*colBlockDim)** be number of columns of **A**, then **A** and **C** are **m*n** sparse matrices. The general BSR format of **A** contains **nnzb(=bsrRowPtrA[mb] - bsrRowPtrA[0])** non-zero blocks, whereas sparse matrix **A** contains **nnz(=nnzb*rowBlockDim*colBlockDim)** elements. The user must allocate enough space for arrays **csrRowPtrC**, **csrColIndC**, and **csrValC**. The requirements are as follows:

csrRowPtrC of **m+1** elements

csrValC of **nnz** elements

csrColIndC of **nnz** elements

The general procedure is as follows:

```
// Given general BSR format (bsrRowPtrA, bsrColIndA, bsrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int m = mb*rowBlockDim;
int n = nb*colBlockDim;
int nnzb = bsrRowPtrA[mb] - bsrRowPtrA[0]; // number of blocks
int nnz  = nnzb * rowBlockDim * colBlockDim; // number of elements
cudaMalloc((void**)&csrRowPtrC, sizeof(int)*(m+1));
cudaMalloc((void**)&csrColIndC, sizeof(int)*nnz);
cudaMalloc((void**)&csrValC, sizeof(float)*nnz);
cusparseSgebsr2csr(handle, dir, mb, nb,
        descrA,
        bsrValA, bsrRowPtrA, bsrColIndA,
        rowBlockDim, colBlockDim,
        descrC,
        csrValC, csrRowPtrC, csrColIndC);
```

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
mb	number of block rows of sparse matrix **A**.
nb	number of block columns of sparse matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
bsrValA	<type> array of **nnzb*rowBlockDim*colBlockDim** non-zero elements of matrix **A**.

bsrRowPtrA	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **A**.
bsrColIndA	integer array of **nnzb** column indices of the non-zero blocks of matrix **A**.
rowBlockDim	number of rows within a block of **A**.
colBlockDim	number of columns within a block of **A**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.

Output

csrValC	<type> array of **nnz** non-zero elements of matrix **C**.
csrRowPtrC	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one of matrix **C**.
csrColIndC	integer array of **nnz** column indices of the non-zero elements of matrix **C**.

See `cusparseStatus_t` for the description of the return status

13.5. cusparse<t>csr2gebsr()

```
cusparseStatus_t
cusparseScsr2gebsr_bufferSize(cusparseHandle_t         handle,
                              cusparseDirection_t      dir,
                              int                      m,
                              int                      n,
                              const cusparseMatDescr_t descrA,
                              const float*             csrValA,
                              const int*               csrRowPtrA,
                              const int*               csrColIndA,
                              int                      rowBlockDim,
                              int                      colBlockDim,
                              int*                     pBufferSize)

cusparseStatus_t
cusparseDcsr2gebsr_bufferSize(cusparseHandle_t         handle,
                              cusparseDirection_t      dir,
                              int                      m,
                              int                      n,
                              const cusparseMatDescr_t descrA,
                              const double*            csrValA,
                              const int*               csrRowPtrA,
                              const int*               csrColIndA,
                              int                      rowBlockDim,
                              int                      colBlockDim,
                              int*                     pBufferSize)

cusparseStatus_t
cusparseCcsr2gebsr_bufferSize(cusparseHandle_t         handle,
                              cusparseDirection_t      dir,
                              int                      m,
                              int                      n,
                              const cusparseMatDescr_t descrA,
                              const cuComplex*         csrValA,
                              const int*               csrRowPtrA,
                              const int*               csrColIndA,
                              int                      rowBlockDim,
                              int                      colBlockDim,
                              int*                     pBufferSize)

cusparseStatus_t
cusparseZcsr2gebsr_bufferSize(cusparseHandle_t         handle,
                              cusparseDirection_t      dir,
                              int                      m,
                              int                      n,
                              const cusparseMatDescr_t descrA,
                              const cuDoubleComplex*   csrValA,
                              const int*               csrRowPtrA,
                              const int*               csrColIndA,
                              int                      rowBlockDim,
                              int                      colBlockDim,
                              int*                     pBufferSize)
```

```
cusparseStatus_t
cusparseXcsr2gebsrNnz(cusparseHandle_t         handle,
                      cusparseDirection_t      dir,
                      int                      m,
                      int                      n,
                      const cusparseMatDescr_t descrA,
                      const int*               csrRowPtrA,
                      const int*               csrColIndA,
                      const cusparseMatDescr_t descrC,
                      int*                     bsrRowPtrC,
```

This function converts a sparse matrix **A** in CSR format (that is defined by arrays **csrValA**, **csrRowPtrA**, and **csrColIndA**) into a sparse matrix **C** in general BSR format (that is defined by the three arrays **bsrValC**, **bsrRowPtrC**, and **bsrColIndC**).

The matrix A is a **m*n** sparse matrix and matrix **C** is a **(mb*rowBlockDim)*(nb*colBlockDim)** sparse matrix, where **mb(=(m+rowBlockDim-1)/rowBlockDim)** is the number of block rows of **C**, and **nb(=(n+colBlockDim-1)/colBlockDim)** is the number of block columns of **C**.

The block of **C** is of size **rowBlockDim*colBlockDim**. If **m** is not multiple of **rowBlockDim** or **n** is not multiple of **colBlockDim**, zeros are filled in.

The implementation adopts a two-step approach to do the conversion. First, the user allocates **bsrRowPtrC** of **mb+1** elements and uses function **cusparseXcsr2gebsrNnz()** to determine the number of nonzero block columns per block row. Second, the user gathers **nnzb** (number of nonzero block columns of matrix **C**) from either **(nnzb=*nnzTotalDevHostPtr)** or **(nnzb=bsrRowPtrC[mb]-bsrRowPtrC[0])** and allocates **bsrValC** of **nnzb*rowBlockDim*colBlockDim** elements and **bsrColIndC** of **nnzb** integers. Finally function **cusparse[S|D|C|Z]csr2gebsr()** is called to complete the conversion.

The user must obtain the size of the buffer required by **csr2gebsr()** by calling **csr2gebsr_bufferSize()**, allocate the buffer, and pass the buffer pointer to **csr2gebsr()**.

The general procedure is as follows:

```
// Given CSR format (csrRowPtrA, csrColIndA, csrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzb;
int mb = (m + rowBlockDim-1)/rowBlockDim;
int nb = (n + colBlockDim-1)/colBlockDim;
int bufferSize;
void *pBuffer;
cusparseScsr2gebsr_bufferSize(handle, dir, m, n,
    descrA, csrValA, csrRowPtrA, csrColIndA,
    rowBlockDim, colBlockDim,
    &bufferSize);
cudaMalloc((void**)&pBuffer, bufferSize);
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzb;
cusparseXcsr2gebsrNnz(handle, dir, m, n,
    descrA, csrRowPtrA, csrColIndA,
    descrC, bsrRowPtrC, rowBlockDim, colBlockDim,
    nnzTotalDevHostPtr,
    pBuffer);
if (NULL != nnzTotalDevHostPtr){
    nnzb = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzb, bsrRowPtrC+mb, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzb -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(rowBlockDim*colBlockDim)*nnzb);
cusparseScsr2gebsr(handle, dir, m, n,
        descrA,
        csrValA, csrRowPtrA, csrColIndA,
        descrC,
        bsrValC, bsrRowPtrC, bsrColIndC,
        rowBlockDim, colBlockDim,
        pBuffer);
```

The routine **cusparseXcsr2gebsrNnz()** has the following properties:

- The routine requires no extra storage
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

The routine **cusparse<t>csr2gebsr()** has the following properties:

- The routine requires no extra storage if **pBuffer != NULL**
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
m	number of rows of sparse matrix **A**.
n	number of columns of sparse matrix **A**.

descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** nonzero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one of matrix **A**.
csrColIndA	integer array of **nnz** column indices of the nonzero elements of matrix **A**.
descrC	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
rowBlockDim	number of rows within a block of **C**.
colBlockDim	number of columns within a block of **C**.
pBuffer	buffer allocated by the user, the size is return by **csr2gebsr_bufferSize()**.

Output

bsrValC	<type> array of **nnzb*rowBlockDim*colBlockDim** nonzero elements of matrix **C**.
bsrRowPtrC	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **C**.
bsrColIndC	integer array of **nnzb** column indices of the nonzero blocks of matrix **C**.
nnzTotalDevHostPtr	total number of nonzero blocks of matrix **C**. Pointer **nnzTotalDevHostPtr** can point to a device memory or host memory.

See `cusparseStatus_t` for the description of the return status

13.6. cusparse<t>coo2csr()

```
cusparseStatus_t
cusparseXcoo2csr(cusparseHandle_t   handle,
                 const int*         cooRowInd,
                 int                nnz,
                 int                m,
                 int*               csrRowPtr,
                 cusparseIndexBase_t idxBase)
```

This function converts the array containing the uncompressed row indices (corresponding to COO format) into an array of compressed row pointers (corresponding to CSR format).

It can also be used to convert the array containing the uncompressed column indices (corresponding to COO format) into an array of column pointers (corresponding to CSC format).

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
cooRowInd	integer array of **nnz** uncompressed row indices.
nnz	number of non-zeros of the sparse matrix (that is also the length of array **cooRowInd**).
m	number of rows of matrix **A**.
idxBase	**CUSPARSE_INDEX_BASE_ZERO** or **CUSPARSE_INDEX_BASE_ONE**.

Output

csrRowPtr	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.

See cusparseStatus_t for the description of the return status

13.7. cusparse<t>csc2dense()

```
cusparseStatus_t
cusparseScsc2dense(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const float*             cscValA,
                   const int*               cscRowIndA,
                   const int*               cscColPtrA,
                   float*                   A,
                   int                      lda)

cusparseStatus_t
cusparseDcsc2dense(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const double*            cscValA,
                   const int*               cscRowIndA,
                   const int*               cscColPtrA,
                   double*                  A,
                   int                      lda)

cusparseStatus_t
cusparseCcsc2dense(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const cuComplex*         cscValA,
                   const int*               cscRowIndA,
                   const int*               cscColPtrA,
                   cuComplex*               A,
                   int                      lda)

cusparseStatus_t
cusparseZcsc2dense(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex*   cscValA,
                   const int*               cscRowIndA,
                   const int*               cscColPtrA,
                   cuDoubleComplex*         A,
                   int                      lda)
```

This function converts the sparse matrix in CSC format that is defined by the three arrays **cscValA**, **cscColPtrA**, and **cscRowIndA** into the matrix **A** in dense format. The dense matrix **A** is filled in with the values of the sparse matrix and with zeros elsewhere.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
cscValA	<type> array of **nnz** (= **cscColPtrA(m)** - **cscColPtrA(0)**) nonzero elements of matrix **A**.
cscRowIndA	integer array of **nnz** (= **cscColPtrA(m)** - **cscColPtrA(0)**) row indices of the nonzero elements of matrix **A**.
cscColPtrA	integer array of **n+1** elements that contains the start of every row and the end of the last column plus one.
lda	leading dimension of dense array **A**.

Output

A	array of dimensions **(lda, n)** that is filled in with the values of the sparse matrix.

See `cusparseStatus_t` for the description of the return status

13.8. cusparse<t>csr2bsr()

```
cusparseStatus_t
cusparseXcsr2bsrNnz(cusparseHandle_t         handle,
                    cusparseDirection_t      dir,
                    int                      m,
                    int                      n,
                    const cusparseMatDescr_t descrA,
                    const int*               csrRowPtrA,
                    const int*               csrColIndA,
                    int                      blockDim,
                    const cusparseMatDescr_t descrC,
                    int*                     bsrRowPtrC,
                    int*                     nnzTotalDevHostPtr)

cusparseStatus_t
cusparseScsr2bsr(cusparseHandle_t         handle,
                 cusparseDirection_t      dir,
                 int                      m,
                 int                      n,
                 const cusparseMatDescr_t descrA,
                 const float*             csrValA,
                 const int*               csrRowPtrA,
                 const int*               csrColIndA,
                 int                      blockDim,
                 const cusparseMatDescr_t descrC,
                 float*                   bsrValC,
                 int*                     bsrRowPtrC,
                 int*                     bsrColIndC)

cusparseStatus_t
cusparseDcsr2bsr(cusparseHandle_t         handle,
                 cusparseDirection_t      dir,
                 int                      m,
                 int                      n,
                 const cusparseMatDescr_t descrA,
                 const double*            csrValA,
                 const int*               csrRowPtrA,
                 const int*               csrColIndA,
                 int                      blockDim,
                 const cusparseMatDescr_t descrC,
                 double*                  bsrValC,
                 int*                     bsrRowPtrC,
                 int*                     bsrColIndC)

cusparseStatus_t
cusparseCcsr2bsr(cusparseHandle_t         handle,
                 cusparseDirection_t      dir,
                 int                      m,
                 int                      n,
                 const cusparseMatDescr_t descrA,
                 const cuComplex*         csrValA,
                 const int*               csrRowPtrA,
                 const int*               csrColIndA,
                 int                      blockDim,
                 const cusparseMatDescr_t descrC,
                 cuComplex*               bsrValC,
                 int*                     bsrRowPtrC,
                 int*                     bsrColIndC)

cusparseStatus_t
cusparseZcsr2bsr(cusparseHandle_t         handle,
                 cusparseDirection_t      dir,
                 int                      m,
                 int                      n,
```

This function converts a sparse matrix in CSR format that is defined by the three arrays **csrValA**, **csrRowPtrA**, and **csrColIndA** into a sparse matrix in BSR format that is defined by arrays **bsrValC**, **bsrRowPtrC**, and **bsrColIndC**.

A is an **m*n** sparse matrix. The BSR format of **A** has **mb** block rows, **nb** block columns, and **nnzb** nonzero blocks, where **mb=((m+blockDim-1)/blockDim)** and **nb=(n +blockDim-1)/blockDim**.

If **m** or **n** is not multiple of **blockDim**, zeros are filled in.

The conversion in cuSPARSE entails a two-step approach. First, the user allocates **bsrRowPtrC** of **mb+1** elements and uses function **cusparseXcsr2bsrNnz()** to determine the number of nonzero block columns per block row. Second, the user gathers **nnzb** (number of non-zero block columns of matrix **C**) from either **(nnzb=*nnzTotalDevHostPtr)** or **(nnzb=bsrRowPtrC[mb]-bsrRowPtrC[0])** and allocates **bsrValC** of **nnzb*blockDim*blockDim** elements and **bsrColIndC** of **nnzb** elements. Finally function **cusparse[S|D|C|Z]csr2bsr90** is called to complete the conversion.

The general procedure is as follows:

```
// Given CSR format (csrRowPtrA, csrcolIndA, csrValA) and
// blocks of BSR format are stored in column-major order.
cusparseDirection_t dir = CUSPARSE_DIRECTION_COLUMN;
int base, nnzb;
int mb = (m + blockDim-1)/blockDim;
cudaMalloc((void**)&bsrRowPtrC, sizeof(int) *(mb+1));
// nnzTotalDevHostPtr points to host memory
int *nnzTotalDevHostPtr = &nnzb;
cusparseXcsr2bsrNnz(handle, dir, m, n,
        descrA, csrRowPtrA, csrColIndA,
        blockDim,
        descrC, bsrRowPtrC,
        nnzTotalDevHostPtr);
if (NULL != nnzTotalDevHostPtr){
    nnzb = *nnzTotalDevHostPtr;
}else{
    cudaMemcpy(&nnzb, bsrRowPtrC+mb, sizeof(int), cudaMemcpyDeviceToHost);
    cudaMemcpy(&base, bsrRowPtrC, sizeof(int), cudaMemcpyDeviceToHost);
    nnzb -= base;
}
cudaMalloc((void**)&bsrColIndC, sizeof(int)*nnzb);
cudaMalloc((void**)&bsrValC, sizeof(float)*(blockDim*blockDim)*nnzb);
cusparseScsr2bsr(handle, dir, m, n,
        descrA,
        csrValA, csrRowPtrA, csrColIndA,
        blockDim,
        descrC,
        bsrValC, bsrRowPtrC, bsrColIndC);
```

The routine **cusparse<t>csr2bsr()** has the following properties:

- This function requires temporary extra storage that is allocated internally if **blockDim > 16**
- The routine does ***not*** support asynchronous execution if **blockDim == 1**
- The routine does ***not*** support CUDA graph capture if **blockDim == 1**

The routine **cusparseXcsr2bsrNnz()** has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution

- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
dir	storage format of blocks, either **CUSPARSE_DIRECTION_ROW** or **CUSPARSE_DIRECTION_COLUMN**.
m	number of rows of sparse matrix **A**.
n	number of columns of sparse matrix **A**.
descrA	the descriptor of matrix **A**.
csrValA	<type> array of **nnz(=csrRowPtrA[m]-csrRowPtr[0])** non-zero elements of matrix **A**.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	integer array of **nnz** column indices of the non-zero elements of matrix **A**.
blockDim	block dimension of sparse matrix **A**. The range of **blockDim** is between 1 and **min(m,n)**.
descrC	the descriptor of matrix **C**.

Output

bsrValC	<type> array of **nnzb*blockDim*blockDim** nonzero elements of matrix **C**.
bsrRowPtrC	integer array of **mb+1** elements that contains the start of every block row and the end of the last block row plus one of matrix **C**.
bsrColIndC	integer array of **nnzb** column indices of the non-zero blocks of matrix **C**.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory. It is equal to **(bsrRowPtrC[mb]-bsrRowPtrC[0])**.

See `cusparseStatus_t` for the description of the return status

13.9. cusparse<t>csr2coo()

```
cusparseStatus_t
cusparseXcsr2coo(cusparseHandle_t    handle,
                 const int*          csrRowPtr,
                 int                 nnz,
                 int                 m,
                 int*                cooRowInd,
                 cusparseIndexBase_t idxBase)
```

This function converts the array containing the compressed row pointers (corresponding to CSR format) into an array of uncompressed row indices (corresponding to COO format).

It can also be used to convert the array containing the compressed column indices (corresponding to CSC format) into an array of uncompressed column indices (corresponding to COO format).

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`csrRowPtr`	integer array of **`m+1`** elements that contains the start of every row and the end of the last row plus one.
`nnz`	number of nonzeros of the sparse matrix (that is also the length of array **`cooRowInd`**).
`m`	number of rows of matrix **`A`**.
`idxBase`	**`CUSPARSE_INDEX_BASE_ZERO`** or **`CUSPARSE_INDEX_BASE_ONE`**.

Output

`cooRowInd`	integer array of **`nnz`** uncompressed row indices.

See `cusparseStatus_t` for the description of the return status

13.10. cusparseCsr2cscEx2()

```
cusparseStatus_t
cusparseCsr2cscEx2_bufferSize(cusparseHandle_t     handle,
                              int                  m,
                              int                  n,
                              int                  nnz,
                              const void*          csrVal,
                              const int*           csrRowPtr,
                              const int*           csrColInd,
                              void*                cscVal,
                              int*                 cscColPtr,
                              int*                 cscRowInd,
                              cudaDataType         valType,
                              cusparseAction_t     copyValues,
                              cusparseIndexBase_t  idxBase,
                              cusparseCsr2CscAlg_t alg,
                              size_t*              bufferSize)
```

```
cusparseStatus_t
cusparseCsr2cscEx2(cusparseHandle_t     handle,
                   int                  m,
                   int                  n,
                   int                  nnz,
                   const void*          csrVal,
                   const int*           csrRowPtr,
                   const int*           csrColInd,
                   void*                cscVal,
                   int*                 cscColPtr,
                   int*                 cscRowInd,
                   cudaDataType         valType,
                   cusparseAction_t     copyValues,
                   cusparseIndexBase_t  idxBase,
                   cusparseCsr2CscAlg_t alg,
                   void*                buffer)
```

This function converts a sparse matrix in CSR format (that is defined by the three arrays **csrVal**, **csrRowPtr**, and **csrColInd**) into a sparse matrix in CSC format (that is defined by arrays **cscVal**, **cscRowInd**, and **cscColPtr**). The resulting matrix can also be seen as the transpose of the original sparse matrix. Notice that this routine can also be used to convert a matrix in CSC format into a matrix in CSR format.

For alg **CUSPARSE_CSR2CSC_ALG1**: it requires extra storage proportional to the number of nonzero values **nnz**. It provides in output always the same matrix.

For alg **CUSPARSE_CSR2CSC_ALG2**: it requires extra storage proportional to the number of rows **m**. It does not ensure always the same ordering of CSC column indices and values. Also, it provides better performance then **CUSPARSE_CSR2CSC_ALG1** for regular matrices.

It is executed asynchronously with respect to the host, and it may return control to the application on the host before the result is ready.

The function **`cusparseCsr2cscEx2_bufferSize()`** returns the size of the workspace needed by **`cusparseCsr2cscEx2()`**. User needs to allocate a buffer of this size and give that buffer to **`cusparseCsr2cscEx2()`** as an argument.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context
`m`	number of rows of the CSR input matrix; number of columns of the CSC ouput matrix
`n`	number of columns of the CSR input matrix; number of rows of the CSC ouput matrix
`nnz`	number of nonzero elements of the CSR and CSC matrices
`csrVal`	value array of size **`nnz`** of the CSR matrix; of same type as **`valType`**
`csrRowPtr`	integer array of size **`m + 1`** that containes the CSR row offsets
`csrColInd`	integer array of size **`nnz`** that containes the CSR column indices
valType	value type for both CSR and CSC matrices
copyValues	**`CUSPARSE_ACTION_SYMBOLIC`** or **`CUSPARSE_ACTION_NUMERIC`**
idxBase	Index base **`CUSPARSE_INDEX_BASE_ZERO`** or **`CUSPARSE_INDEX_BASE_ONE`**.
alg	algorithm implementation. see **`cusparseCsr2CscAlg_t`** for possible values.
`bufferSize`	number of bytes of workspace needed by **`cusparseCsr2cscEx2()`**
buffer	pointer to workspace buffer

See `cusparseStatus_t` for the description of the return status

13.11. cusparse<t>csr2dense()

```
cusparseStatus_t
cusparseScsr2dense(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const float*             csrValA,
                   const int*               csrRowPtrA,
                   const int*               csrColIndA,
                   float*                   A,
                   int                      lda)

cusparseStatus_t
cusparseDcsr2dense(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const double*            csrValA,
                   const int*               csrRowPtrA,
                   const int*               csrColIndA,
                   double*                  A,
                   int                      lda)

cusparseStatus_t
cusparseCcsr2dense(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const cuComplex*         csrValA,
                   const int*               csrRowPtrA,
                   const int*               csrColIndA,
                   cuComplex*               A,
                   int                      lda)

cusparseStatus_t
cusparseZcsr2dense(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex*   csrValA,
                   const int*               csrRowPtrA,
                   const int*               csrColIndA,
                   cuDoubleComplex*         A,
                   int                      lda)
```

This function converts the sparse matrix in CSR format (that is defined by the three arrays **`csrValA`**, **`csrRowPtrA`**, and **`csrColIndA`**) into the matrix **`A`** in dense format. The dense matrix **`A`** is filled in with the values of the sparse matrix and with zeros elsewhere.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows of matrix A.
`n`	number of columns of matrix A.
`descrA`	the descriptor of matrix A. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL`. Also, the supported index bases are `CUSPARSE_INDEX_BASE_ZERO` and `CUSPARSE_INDEX_BASE_ONE`.
`csrValA`	<type> array of `nnz` ($=$ `csrRowPtrA(m)` - `csrRowPtrA(0)`) nonzero elements of matrix A.
`csrRowPtrA`	integer array of `m+1` elements that contains the start of every row and the end of the last row plus one.
`csrColIndA`	integer array of `nnz` ($=$ `csrRowPtrA(m)` - `csrRowPtrA(0)`) column indices of the nonzero elements of matrix A.
`lda`	leading dimension of array matrix`A`.

Output

`A`	array of dimensions `(lda,n)` that is filled in with the values of the sparse matrix.

See `cusparseStatus_t` for the description of the return status

13.12. cusparse<t>csr2csr_compress()

```
cusparseStatus_t
cusparseScsr2csr_compress(cusparseHandle_t         handle,
                          int                      m,
                          int                      n,
                          const cusparseMatDescr_t descrA,
                          const float*             csrValA,
                          const int*               csrColIndA,
                          const int*               csrRowPtrA,
                          int                      nnzA,
                          const int*               nnzPerRow,
                          float*                   csrValC,
                          int*                     csrColIndC,
                          int*                     csrRowPtrC,
                          float                    tol)

cusparseStatus_t
cusparseDcsr2csr_compress(cusparseHandle_t         handle,
                          int                      m,
                          int                      n,
                          const cusparseMatDescr_t descrA,
                          const double*            csrValA,
                          const int*               csrColIndA,
                          const int*               csrRowPtrA,
                          int                      nnzA,
                          const int*               nnzPerRow,
                          double*                  csrValC,
                          int*                     csrColIndC,
                          int*                     csrRowPtrC,
                          double                   tol)

cusparseStatus_t
cusparseCcsr2csr_compress(cusparseHandle_t         handle,
                          int                      m,
                          int                      n,
                          const cusparseMatDescr_t descrA,
                          const cuComplex*         csrValA,
                          const int*               csrColIndA,
                          const int*               csrRowPtrA,
                          int                      nnzA,
                          const int*               nnzPerRow,
                          cuComplex*               csrValC,
                          int*                     csrColIndC,
                          int*                     csrRowPtrC,
                          cuComplex                tol)

cusparseStatus_t
cusparseZcsr2csr_compress(cusparseHandle_t         handle,
                          int                      m,
                          int                      n,
                          const cusparseMatDescr_t descrA,
                          const cuDoubleComplex*   csrValA,
                          const int*               csrColIndA,
                          const int*               csrRowPtrA,
                          int                      nnzA,
                          const int*               nnzPerRow,
                          cuDoubleComplex*         csrValC,
                          int*                     csrColIndC,
                          int*                     csrRowPtrC,
                          cuDoubleComplex          tol)
```

This function compresses the sparse matrix in CSR format into compressed CSR format. Given a sparse matrix A and a non-negative value threshold(in the case of complex values, only the magnitude of the real part is used in the check), the function returns a sparse matrix C, defined by

$$C(i,j) = A(i,j) \quad \text{if } |A(i,j)| > \text{threshold}$$

The implementation adopts a two-step approach to do the conversion. First, the user allocates csrRowPtrC of m+1 elements and uses function cusparse<t>nnz_compress() to determine nnzPerRow(the number of nonzeros columns per row) and nnzC(the total number of nonzeros). Second, the user allocates csrValC of nnzC elements and csrColIndC of nnzC integers. Finally function cusparse<t>csr2csr_compress() is called to complete the conversion.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

handle	handle to the cuSPARSE library context.
m	number of rows of matrix A.
n	number of columns of matrix A.
descrA	the descriptor of matrix A. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	<type> array of **nnz** ($=$ **csrRowPtrA(m)** $-$ **csrRowPtrA(0)**) elements of matrix A.
csrColIndA	integer array of **nnz** ($=$ **csrRowPtrA(m)** $-$ **csrRowPtrA(0)**) column indices of the elements of matrix A.
csrRowPtrA	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
nnzA	number of nonzero elements in matrix A.
nnzPerRow	this array contains the number of elements kept in the compressed matrix, by row.
tol	on input, this contains the non-negative tolerance value used for compression. Any values in matrix A less than or equal to this value will be dropped during compression.

Output

csrValC	on output, this array contains the typed values of elements kept in the compressed matrix. Size = nnzC.

csrColIndC	on output, this integer array contains the column indices of elements kept in the compressed matrix. Size = nnzC.
csrRowPtrC	on output, this integer array contains the row pointers for elements kept in the compressed matrix. Size = m+1

See `cusparseStatus_t` for the description of the return status

The following is a sample code to show how to use this API.

```
#include <stdio.h>
#include <sys/time.h>
#include <cusparse.h>

#define ERR_NE(X,Y) do { if ((X) != (Y)) { \
                             fprintf(stderr,"Error in %s at %s:%d
\n",__func__,__FILE__,__LINE__); \
                             exit(-1);}} while(0)
#define CUDA_CALL(X) ERR_NE((X),cudaSuccess)
#define CUSPARSE_CALL(X) ERR_NE((X),CUSPARSE_STATUS_SUCCESS)
int main(){
    int m = 6, n = 5;
    cusparseHandle_t  handle;
    CUSPARSE_CALL( cusparseCreate(&handle) );
    cusparseMatDescr_t descrX;
    CUSPARSE_CALL(cusparseCreateMatDescr(&descrX));
    // Initialize sparse matrix
    float *X;
    CUDA_CALL(cudaMallocManaged( &X, sizeof(float) * m * n ));
    memset( X, 0, sizeof(float) * m * n );
    X[0 + 0*m] = 1.0;  X[0 + 1*m] = 3.0;
    X[1 + 1*m] = -4.0; X[1 + 2*m] = 5.0;
    X[2 + 0*m] = 2.0;  X[2 + 3*m] = 7.0;  X[2 + 4*m] = 8.0;
    X[3 + 2*m] = 6.0;  X[3 + 4*m] = 9.0;
    X[4 + 3*m] = 3.5;  X[4 + 4*m] = 5.5;
    X[5 + 0*m] = 6.5;  X[5 + 2*m] = -9.9;
    // Initialize total_nnz, and  nnzPerRowX for cusparseSdense2csr()
    int total_nnz = 13;
    int *nnzPerRowX;
    CUDA_CALL( cudaMallocManaged( &nnzPerRowX, sizeof(int) * m ));
    nnzPerRowX[0] = 2;  nnzPerRowX[1] = 2;  nnzPerRowX[2] = 3;
    nnzPerRowX[3] = 2;  nnzPerRowX[4] = 2;  nnzPerRowX[5] = 2;

    float *csrValX;
    int *csrRowPtrX;
    int *csrColIndX;
    CUDA_CALL( cudaMallocManaged( &csrValX, sizeof(float) * total_nnz) );
    CUDA_CALL( cudaMallocManaged( &csrRowPtrX, sizeof(int) * (m+1))) ;
    CUDA_CALL( cudaMallocManaged( &csrColIndX, sizeof(int) * total_nnz)) ;
```

Before calling this API, call two APIs to prepare the input.

```
/** Call cusparseSdense2csr to generate CSR format as the inputs for
    cusparseScsr2csr_compress  **/
    CUSPARSE_CALL( cusparseSdense2csr( handle, m, n, descrX, X,
                                       m, nnzPerRowX, csrValX,
                                       csrRowPtrX, csrColIndX )) ;
    float tol = 3.5;
    int *nnzPerRowY;
    int *testNNZTotal;
    CUDA_CALL (cudaMallocManaged( &nnzPerRowY, sizeof(int) * m ));
    CUDA_CALL (cudaMallocManaged( &testNNZTotal, sizeof(int)));
    memset( nnzPerRowY, 0, sizeof(int) * m );
    // cusparseSnnz_compress generates nnzPerRowY and testNNZTotal
    CUSPARSE_CALL( cusparseSnnz_compress(handle, m, descrX, csrValX,
                                         csrRowPtrX, nnzPerRowY,
                                         testNNZTotal, tol));

    float *csrValY;
    int *csrRowPtrY;
    int *csrColIndY;
    CUDA_CALL( cudaMallocManaged( &csrValY, sizeof(float) * (*testNNZTotal)));
    CUDA_CALL( cudaMallocManaged( &csrRowPtrY, sizeof(int) * (m+1)));
    CUDA_CALL( cudaMallocManaged( &csrColIndY, sizeof(int) * (*testNNZTotal)));

    CUSPARSE_CALL( cusparseScsr2csr_compress( handle, m, n, descrX, csrValX,
                                              csrColIndX, csrRowPtrX,
                                              total_nnz,  nnzPerRowY,
                                              csrValY, csrColIndY,
                                              csrRowPtrY, tol));
    /* Expect results
    nnzPerRowY:  0 2 2 2 1 2
    csrValY:     -4 5 7 8 6 9 5.5 6.5 -9.9
    csrColIndY:  1 2 3 4 2 4 4 0 2
    csrRowPtrY:  0 0 2 4 6 7 9
    */
    cudaFree(X);
    cusparseDestroy(handle);
    cudaFree(nnzPerRowX);
    cudaFree(csrValX);
    cudaFree(csrRowPtrX);
    cudaFree(csrColIndX);
    cudaFree(csrValY);
    cudaFree(nnzPerRowY);
    cudaFree(testNNZTotal);
    cudaFree(csrRowPtrY);
    cudaFree(csrColIndY);
    return 0;
}
```

13.13. cusparse<t>dense2csc()

```
cusparseStatus_t
cusparseSdense2csc(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const float*             A,
                   int                      lda,
                   const int*               nnzPerCol,
                   float*                   cscValA,
                   int*                     cscRowIndA,
                   int*                     cscColPtrA)

cusparseStatus_t
cusparseDdense2csc(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const double*            A,
                   int                      lda,
                   const int*               nnzPerCol,
                   double*                  cscValA,
                   int*                     cscRowIndA,
                   int*                     cscColPtrA)

cusparseStatus_t
cusparseCdense2csc(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const cuComplex*         A,
                   int                      lda,
                   const int*               nnzPerCol,
                   cuComplex*               cscValA,
                   int*                     cscRowIndA,
                   int*                     cscColPtrA)

cusparseStatus_t
cusparseZdense2csc(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex*   A,
                   int                      lda,
                   const int*               nnzPerCol,
                   cuDoubleComplex*         cscValA,
                   int*                     cscRowIndA,
                   int*                     cscColPtrA)
```

This function converts the matrix **A** in dense format into a sparse matrix in CSC format. All the parameters are assumed to have been pre-allocated by the user, and the arrays are filled in based on **nnzPerCol**, which can be precomputed with **cusparse<t>nnz()**.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution

- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows of matrix **`A`**.
`n`	number of columns of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`**.
`A`	array of dimensions **`(lda, n)`**.
`lda`	leading dimension of dense array **`A`**.
`nnzPerCol`	array of size **`n`** containing the number of nonzero elements per column.

Output

`cscValA`	<type> array of **`nnz`** (= **`cscRowPtrA(m)`** - **`cscRowPtrA(0)`**) nonzero elements of matrix **`A`**. It is only filled in if **`copyValues`** is set to **`CUSPARSE_ACTION_NUMERIC`**.
`cscRowIndA`	integer array of **`nnz`** (= **`cscRowPtrA(m)`** - **`cscRowPtrA(0)`**) row indices of the nonzero elements of matrix **`A`**.
`cscColPtrA`	integer array of **`n+1`** elements that contains the start of every column and the end of the last column plus one.

See `cusparseStatus_t` for the description of the return status

13.14. cusparse<t>dense2csr()

```
cusparseStatus_t
cusparseSdense2csr(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const float*             A,
                   int                      lda,
                   const int*               nnzPerRow,
                   float*                   csrValA,
                   int*                     csrRowPtrA,
                   int*                     csrColIndA)

cusparseStatus_t
cusparseDdense2csr(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const double*            A,
                   int                      lda,
                   const int*               nnzPerRow,
                   double*                  csrValA,
                   int*                     csrRowPtrA,
                   int*                     csrColIndA)

cusparseStatus_t
cusparseCdense2csr(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const cuComplex*         A,
                   int                      lda,
                   const int*               nnzPerRow,
                   cuComplex*               csrValA,
                   int*                     csrRowPtrA,
                   int*                     csrColIndA)

cusparseStatus_t
cusparseZdense2csr(cusparseHandle_t         handle,
                   int                      m,
                   int                      n,
                   const cusparseMatDescr_t descrA,
                   const cuDoubleComplex*   A,
                   int                      lda,
                   const int*               nnzPerRow,
                   cuDoubleComplex*         csrValA,
                   int*                     csrRowPtrA,
                   int*                     csrColIndA)
```

This function converts the matrix **`A`** in dense format into a sparse matrix in CSR format. All the parameters are assumed to have been pre-allocated by the user and the arrays are filled in based on **`nnzPerRow`**, which can be pre-computed with **`cusparse<t>nnz()`**.

This function requires no extra storage. It is executed asynchronously with respect to the host and may return control to the application on the host before the result is ready.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows of matrix `A`.
`n`	number of columns of matrix `A`.
`descrA`	the descriptor of matrix `A`. The supported matrix type is `CUSPARSE_MATRIX_TYPE_GENERAL`. Also, the supported index bases are `CUSPARSE_INDEX_BASE_ZERO` and `CUSPARSE_INDEX_BASE_ONE`.
`A`	array of dimensions `(lda, n)`.
`lda`	leading dimension of dense array `A`.
`nnzPerRow`	array of size `n` containing the number of non-zero elements per row.

Output

`csrValA`	<type> array of `nnz` (= `csrRowPtrA(m)` - `csrRowPtrA(0)`) nonzero elements of matrix `A`.
`csrRowPtrA`	integer array of `m+1` elements that contains the start of every column and the end of the last column plus one.
`csrColIndA`	integer array of `nnz` (= `csrRowPtrA(m)` - `csrRowPtrA(0)`) column indices of the non-zero elements of matrix `A`.

See `cusparseStatus_t` for the description of the return status

13.15. cusparse<t>nnz()

```
cusparseStatus_t
cusparseSnnz(cusparseHandle_t        handle,
             cusparseDirection_t     dirA,
             int                      m,
             int                      n,
             const cusparseMatDescr_t descrA,
             const float*             A,
             int                      lda,
             int*                     nnzPerRowColumn,
             int*                     nnzTotalDevHostPtr)

cusparseStatus_t
cusparseDnnz(cusparseHandle_t        handle,
             cusparseDirection_t     dirA,
             int                      m,
             int                      n,
             const cusparseMatDescr_t descrA,
             const double*            A,
             int                      lda,
             int*                     nnzPerRowColumn,
             int*                     nnzTotalDevHostPtr)

cusparseStatus_t
cusparseCnnz(cusparseHandle_t        handle,
             cusparseDirection_t     dirA,
             int                      m,
             int                      n,
             const cusparseMatDescr_t descrA,
             const cuComplex*         A,
             int                      lda,
             int*                     nnzPerRowColumn,
             int*                     nnzTotalDevHostPtr)

cusparseStatus_t
cusparseZnnz(cusparseHandle_t        handle,
             cusparseDirection_t     dirA,
             int                      m,
             int                      n,
             const cusparseMatDescr_t descrA,
             const cuDoubleComplex*   A,
             int                      lda,
             int*                     nnzPerRowColumn,
             int*                     nnzTotalDevHostPtr)
```

This function computes the number of nonzero elements per row or column and the total number of nonzero elements in a dense matrix.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.

dirA	direction that specifies whether to count nonzero elements by **CUSPARSE_DIRECTION_ROW** or by **CUSPARSE_DIRECTION_COLUMN**.
m	number of rows of matrix **A**.
n	number of columns of matrix **A**.
descrA	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**. Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
A	array of dimensions **(lda, n)**.
lda	leading dimension of dense array **A**.

Output

nnzPerRowColumn	array of size **m** or **n** containing the number of nonzero elements per row or column, respectively.
nnzTotalDevHostPtr	total number of nonzero elements in device or host memory.

See `cusparseStatus_t` for the description of the return status

13.16. cusparseCreateIdentityPermutation()

```
cusparseStatus_t
cusparseCreateIdentityPermutation(cusparseHandle_t handle,
                                  int              n,
                                  int*             p);
```

This function creates an identity map. The output parameter **p** represents such map by **p = 0:1:(n-1)**.

This function is typically used with **coosort**, **csrsort**, **cscsort**.

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

parameter	**device or host**	**description**
handle	**host**	handle to the cuSPARSE library context.
n	**host**	size of the map.

Output

parameter	**device or host**	**description**
p	**device**	integer array of dimensions **n**.

See `cusparseStatus_t` for the description of the return status

13.17. cusparseXcoosort()

```
cusparseStatus_t
cusparseXcoosort_bufferSizeExt(cusparseHandle_t handle,
                               int              m,
                               int              n,
                               int              nnz,
                               const int*       cooRows,
                               const int*       cooCols,
                               size_t*          pBufferSizeInBytes)
```

```
cusparseStatus_t
cusparseXcoosortByRow(cusparseHandle_t handle,
                      int              m,
                      int              n,
                      int              nnz,
                      int*             cooRows,
                      int*             cooCols,
                      int*             P,
                      void*            pBuffer)

cusparseStatus_t
cusparseXcoosortByColumn(cusparseHandle_t handle,
                         int              m,
                         int              n,
                         int              nnz,
                         int*             cooRows,
                         int*             cooCols,
                         int*             P,
                         void*            pBuffer);
```

This function sorts COO format. The sorting is in-place. Also the user can sort by row or sort by column.

A is an **m×n** sparse matrix that is defined in COO storage format by the three arrays **cooVals**, **cooRows**, and **cooCols**.

There is no assumption for the base index of the matrix. **coosort** uses stable sort on signed integer, so the value of **cooRows** or **cooCols** can be negative.

This function **coosort()** requires buffer size returned by **coosort_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The parameter **P** is both input and output. If the user wants to compute sorted **cooVal**, **P** must be set as 0:1:(nnz-1) before **coosort()**, and after **coosort()**, new sorted value array satisfies **cooVal_sorted = cooVal(P)**.

Remark: the dimension **m** and **n** are not used. If the user does not know the value of **m** or **n**, just passes a value positive. This usually happens if the user only reads a COO array first and needs to decide the dimension **m** or **n** later.

Appendix section provides a simple example of **coosort()**.

- The routine requires no extra storage if **`pBuffer != NULL`**
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

parameter	device or host	description
`handle`	**`host`**	handle to the cuSPARSE library context.
`m`	**`host`**	number of rows of matrix **`A`**.
`n`	**`host`**	number of columns of matrix **`A`**.
`nnz`	**`host`**	number of nonzero elements of matrix **`A`**.
`cooRows`	**`device`**	integer array of **`nnz`** unsorted row indices of **`A`**.
`cooCols`	**`device`**	integer array of **`nnz`** unsorted column indices of **`A`**.
`P`	**`device`**	integer array of **`nnz`** unsorted map indices. To construct **`cooVal`**, the user has to set **`P=0:1:(nnz-1)`**.
`pBuffer`	**`device`**	buffer allocated by the user; the size is returned by **`coosort_bufferSizeExt()`**.

Output

parameter	device or host	description
`cooRows`	**`device`**	integer array of **`nnz`** sorted row indices of **`A`**.
`cooCols`	**`device`**	integer array of **`nnz`** sorted column indices of **`A`**.
`P`	**`device`**	integer array of **`nnz`** sorted map indices.
`pBufferSizeInBytes`	**`host`**	number of bytes of the buffer.

See `cusparseStatus_t` for the description of the return status

13.18. cusparseXcsrsort()

```
cusparseStatus_t
cusparseXcsrsort_bufferSizeExt(cusparseHandle_t handle,
                               int              m,
                               int              n,
                               int              nnz,
                               const int*       csrRowPtr,
                               const int*       csrColInd,
                               size_t*          pBufferSizeInBytes)
```

```
cusparseStatus_t
cusparseXcsrsort(cusparseHandle_t         handle,
                 int                      m,
                 int                      n,
                 int                      nnz,
                 const cusparseMatDescr_t descrA,
                 const int*               csrRowPtr,
                 int*                     csrColInd,
                 int*                     P,
                 void*                    pBuffer)
```

This function sorts CSR format. The stable sorting is in-place.

The matrix type is regarded as **CUSPARSE_MATRIX_TYPE_GENERAL** implicitly. In other words, any symmetric property is ignored.

This function **csrsort()** requires buffer size returned by **csrsort_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The parameter **P** is both input and output. If the user wants to compute sorted **csrVal**, **P** must be set as 0:1:(nnz-1) before **csrsort()**, and after **csrsort()**, new sorted value array satisfies **csrVal_sorted = csrVal(P)**.

The general procedure is as follows:

```
// A is a 3x3 sparse matrix, base-0
//     | 1 2 3 |
// A = | 4 5 6 |
//     | 7 8 9 |
const int m = 3;
const int n = 3;
const int nnz = 9;
csrRowPtr[m+1] = { 0, 3, 6, 9}; // on device
csrColInd[nnz] = { 2, 1, 0, 0, 2,1, 1, 2, 0}; // on device
csrVal[nnz] = { 3, 2, 1, 4, 6, 5, 8, 9, 7}; // on device
size_t pBufferSizeInBytes = 0;
void *pBuffer = NULL;
int *P = NULL;

// step 1: allocate buffer
cusparseXcsrsort_bufferSizeExt(handle, m, n, nnz, csrRowPtr, csrColInd,
 &pBufferSizeInBytes);
cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

// step 2: setup permutation vector P to identity
cudaMalloc( (void**)&P, sizeof(int)*nnz);
cusparseCreateIdentityPermutation(handle, nnz, P);

// step 3: sort CSR format
cusparseXcsrsort(handle, m, n, nnz, descrA, csrRowPtr, csrColInd, P, pBuffer);

// step 4: gather sorted csrVal
cusparseDgthr(handle, nnz, csrVal, csrVal_sorted, P, CUSPARSE_INDEX_BASE_ZERO);
```

- The routine requires no extra storage if **`pBuffer != NULL`**
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

parameter	**device or host**	**description**
handle	**host**	handle to the cuSPARSE library context.
m	**host**	number of rows of matrix **A**.
n	**host**	number of columns of matrix **A**.
nnz	**host**	number of nonzero elements of matrix **A**.
csrRowsPtr	**device**	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColInd	**device**	integer array of **nnz** unsorted column indices of **A**.
P	**device**	integer array of **nnz** unsorted map indices. To construct **csrVal**, the user has to set **P=0:1:(nnz-1)**.
pBuffer	**device**	buffer allocated by the user; the size is returned by **csrsort_bufferSizeExt()**.

Output

parameter	**device or host**	**description**
csrColInd	**device**	integer array of **nnz** sorted column indices of **A**.

P	**device**	integer array of **nnz** sorted map indices.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

See cusparseStatus_t for the description of the return status

13.19. cusparseXcscsort()

```
cusparseStatus_t
cusparseXcscsort_bufferSizeExt(cusparseHandle_t handle,
                               int              m,
                               int              n,
                               int              nnz,
                               const int*       cscColPtr,
                               const int*       cscRowInd,
                               size_t*          pBufferSizeInBytes)
```

```
cusparseStatus_t
cusparseXcscsort(cusparseHandle_t         handle,
                 int                      m,
                 int                      n,
                 int                      nnz,
                 const cusparseMatDescr_t descrA,
                 const int*               cscColPtr,
                 int*                     cscRowInd,
                 int*                     P,
                 void*                    pBuffer)
```

This function sorts CSC format. The stable sorting is in-place.

The matrix type is regarded as **CUSPARSE_MATRIX_TYPE_GENERAL** implicitly. In other words, any symmetric property is ignored.

This function **cscsort()** requires buffer size returned by **cscsort_bufferSizeExt()**. The address of **pBuffer** must be multiple of 128 bytes. If not, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

The parameter **P** is both input and output. If the user wants to compute sorted **cscVal**, **P** must be set as 0:1:(nnz-1) before **cscsort()**, and after **cscsort()**, new sorted value array satisfies **cscVal_sorted = cscVal(P)**.

The general procedure is as follows:

```
// A is a 3x3 sparse matrix, base-0
//     | 1 2  |
// A = | 4 0  |
//     | 0 8  |
const int m = 3;
const int n = 2;
const int nnz = 4;
cscColPtr[n+1] = { 0, 2, 4}; // on device
cscRowInd[nnz] = { 1, 0, 2, 0}; // on device
cscVal[nnz]    = { 4.0, 1.0, 8.0, 2.0 }; // on device
size_t pBufferSizeInBytes = 0;
void *pBuffer = NULL;
int *P = NULL;

// step 1: allocate buffer
cusparseXcscsort_bufferSizeExt(handle, m, n, nnz, cscColPtr, cscRowInd,
 &pBufferSizeInBytes);
cudaMalloc( &pBuffer, sizeof(char)* pBufferSizeInBytes);

// step 2: setup permutation vector P to identity
cudaMalloc( (void**)&P, sizeof(int)*nnz);
cusparseCreateIdentityPermutation(handle, nnz, P);

// step 3: sort CSC format
cusparseXcscsort(handle, m, n, nnz, descrA, cscColPtr, cscRowInd, P, pBuffer);

// step 4: gather sorted cscVal
cusparseDgthr(handle, nnz, cscVal, cscVal_sorted, P, CUSPARSE_INDEX_BASE_ZERO);
```

- The routine requires no extra storage if **`pBuffer != NULL`**
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

Input

parameter	device or host	description
`handle`	host	handle to the cuSPARSE library context.
`m`	host	number of rows of matrix `A`.
`n`	host	number of columns of matrix `A`.
`nnz`	host	number of nonzero elements of matrix `A`.
`cscColPtr`	device	integer array of `n+1` elements that contains the start of every column and the end of the last column plus one.
`cscRowInd`	device	integer array of `nnz` unsorted row indices of `A`.
`P`	device	integer array of `nnz` unsorted map indices. To construct `cscVal`, the user has to set `P=0:1:(nnz-1)`.
`pBuffer`	device	buffer allocated by the user; the size is returned by `cscsort_bufferSizeExt()`.

Output

parameter	device or host	description
`cscRowInd`	device	integer array of `nnz` sorted row indices of `A`.

P	**device**	integer array of **nnz** sorted map indices.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

See `cusparseStatus_t` for the description of the return status

13.20. cusparseXcsru2csr()

```
cusparseStatus_t
cusparseCreateCsru2csrInfo(csru2csrInfo_t *info);

cusparseStatus_t
cusparseDestroyCsru2csrInfo(csru2csrInfo_t info);

cusparseStatus_t
cusparseScsru2csr_bufferSizeExt(cusparseHandle_t handle,
                                int              m,
                                int              n,
                                int              nnz,
                                float*           csrVal,
                                const int*       csrRowPtr,
                                int*             csrColInd,
                                csru2csrInfo_t   info,
                                size_t*          pBufferSizeInBytes)

cusparseStatus_t
cusparseDcsru2csr_bufferSizeExt(cusparseHandle_t handle,
                                int              m,
                                int              n,
                                int              nnz,
                                double*          csrVal,
                                const int*       csrRowPtr,
                                int*             csrColInd,
                                csru2csrInfo_t   info,
                                size_t*          pBufferSizeInBytes)

cusparseStatus_t
cusparseCcsru2csr_bufferSizeExt(cusparseHandle_t handle,
                                int              m,
                                int              n,
                                int              nnz,
                                cuComplex*       csrVal,
                                const int*       csrRowPtr,
                                int*             csrColInd,
                                csru2csrInfo_t   info,
                                size_t*          pBufferSizeInBytes)

cusparseStatus_t
cusparseZcsru2csr_bufferSizeExt(cusparseHandle_t handle,
                                int              m,
                                int              n,
                                int              nnz,
                                cuDoubleComplex* csrVal,
                                const int*       csrRowPtr,
                                int*             csrColInd,
                                csru2csrInfo_t   info,
                                size_t*          pBufferSizeInBytes)
```

```
cusparseStatus_t
cusparseScsru2csr(cusparseHandle_t         handle,
                  int                      m,
                  int                      n,
                  int                      nnz,
                  const cusparseMatDescr_t descrA,
                  float*                   csrVal,
                  const int*               csrRowPtr,
                  int*                     csrColInd,
                  csru2csrInfo_t           info,
                  void*                    pBuffer)
```

This function transfers unsorted CSR format to CSR format, and vice versa. The operation is in-place.

This function is a wrapper of **csrsort** and **gthr**. The usecase is the following scenario.

If the user has a matrix **A** of CSR format which is unsorted, and implements his own code (which can be CPU or GPU kernel) based on this special order (for example, diagonal first, then lower triangle, then upper triangle), and wants to convert it to CSR format when calling CUSPARSE library, and then convert it back when doing something else on his/her kernel. For example, suppose the user wants to solve a linear system **Ax=b** by the following iterative scheme

$$x^{(k+1)} = x^{(k)} + L^{(-1)} * (b - Ax^{(k)})$$

The code heavily uses SpMv and triangular solve. Assume that the user has an in-house design of SpMV (Sparse Matrix-Vector multiplication) based on special order of **A**. However the user wants to use CUSAPRSE library for triangular solver. Then the following code can work.

do
step 1: compute residual vector $r = b - A\ x^{(k)}$ by
step 2: B := sort(A), and L is lower triangular part of B
(only sort A once and keep the permutation vector)
step 3: solve $z = L^{(-1)} * (\ b$
step 4: add correction $x^{(k+1)} = x^{(k)} + z$
step 5: A := unsort(B)
(use permutation vector to get back the unsorted CSR)
until convergence

The requirements of step 2 and step 5 are

1. In-place operation.

2. The permutation vector **P** is hidden in an opaque structure.

3. No **cudaMalloc** inside the conversion routine. Instead, the user has to provide the buffer explicitly.

4. The conversion between unsorted CSR and sorted CSR may needs several times, but the function only generates the permutation vector **P** once.

5. The function is based on **csrsort**, **gather** and **scatter** operations.

The operation is called **csru2csr**, which means unsorted CSR to sorted CSR. Also we provide the inverse operation, called **csr2csru**.

In order to keep the permutation vector invisible, we need an opaque structure called **csru2csrInfo**. Then two functions (**cusparseCreateCsru2csrInfo**, **cusparseDestroyCsru2csrInfo**) are used to initialize and to destroy the opaque structure.

cusparse[S|D|C|Z]csru2csr_bufferSizeExt returns the size of the buffer. The permutation vector **P** is also allcated inside **csru2csrInfo**. The lifetime of the permutation vector is the same as the lifetime of **csru2csrInfo**.

cusparse[S|D|C|Z]csru2csr performs forward transformation from unsorted CSR to sorted CSR. First call uses csrsort to generate the permutation vector **P**, and subsequent call uses **P** to do transformation.

cusparse[S|D|C|Z]csr2csru performs backward transformation from sorted CSR to unsorted CSR. **P** is used to get unsorted form back.

The routine **cusparse<t>csru2csr()** has the following properties:

- The routine requires no extra storage if **pBuffer != NULL**
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

The routine **cusparse<t>csr2csru()** has the following properties if **pBuffer != NULL**:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

The following tables describe parameters of **csr2csru_bufferSizeExt** and **csr2csru**.

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.
nnz	host	number of nonzero elements of matrix **A**.
descrA	host	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrVal	device	<type> array of nnz unsorted nonzero elements of matrix **A**.
csrRowsPtr	device	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColInd	device	integer array of **nnz** unsorted column indices of **A**.
info	host	opaque structure initialized using **cusparseCreateCsru2csrInfo()**.
pBuffer	device	buffer allocated by the user; the size is returned by **csru2csr_bufferSizeExt()**.

Output

parameter	device or host	description
csrVal	device	<type> array of nnz sorted nonzero elements of matrix **A**.
csrColInd	device	integer array of **nnz** sorted column indices of **A**.

pBufferSizeInBytes	**host**	number of bytes of the buffer.

See `cusparseStatus_t` for the description of the return status

13.21. cusparseXpruneDense2csr()

```
cusparseStatus_t
cusparseHpruneDense2csr_bufferSizeExt(cusparseHandle_t          handle,
                                       int                       m,
                                       int                       n,
                                       const __half*             A,
                                       int                       lda,
                                       const __half*
 threshold,
                                       const cusparseMatDescr_t descrC,
                                       const __half*             csrValC,
                                       const int*
 csrRowPtrC,
                                       const int*
 csrColIndC,
                                       size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseSpruneDense2csr_bufferSizeExt(cusparseHandle_t          handle,
                                       int                       m,
                                       int                       n,
                                       const float*              A,
                                       int                       lda,
                                       const float*
 threshold,
                                       const cusparseMatDescr_t descrC,
                                       const float*              csrValC,
                                       const int*
 csrRowPtrC,
                                       const int*
 csrColIndC,
                                       size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseDpruneDense2csr_bufferSizeExt(cusparseHandle_t          handle,
                                       int                       m,
                                       int                       n,
                                       const double*             A,
                                       int                       lda,
                                       const double*
 threshold,
                                       const cusparseMatDescr_t descrC,
                                       const double*             csrValC,
                                       const int*
 csrRowPtrC,
                                       const int*
 csrColIndC,
                                       size_t*
 pBufferSizeInBytes)
```

```
cusparseStatus_t
cusparseHpruneDense2csrNnz(cusparseHandle_t          handle,
                           int                       m,
                           int                       n,
                           const __half*             A,
                           int                       lda,
                           const __half*             threshold,
                           const cusparseMatDescr_t descrC,
                           int*                      csrRowPtrC,
                           int*                      nnzTotalDevHostPtr,
                           void*                     pBuffer)
```

This function prunes a dense matrix to a sparse matrix with CSR format.

Given a dense matrix **A** and a non-negative value **threshold**, the function returns a sparse matrix **C**, defined by

$$C(i,j) = A(i,j) \quad \text{if } |A(i,j)| > \text{threshold}$$

The implementation adopts a two-step approach to do the conversion. First, the user allocates **csrRowPtrC** of **m+1** elements and uses function **pruneDense2csrNnz()** to determine the number of nonzeros columns per row. Second, the user gathers **nnzC** (number of nonzeros of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC[m]-csrRowPtrC[0])** and allocates **csrValC** of **nnzC** elements and **csrColIndC** of **nnzC** integers. Finally function **pruneDense2csr()** is called to complete the conversion.

The user must obtain the size of the buffer required by **pruneDense2csr()** by calling **pruneDense2csr_bufferSizeExt()**, allocate the buffer, and pass the buffer pointer to **pruneDense2csr()**.

Appendix section provides a simple example of **pruneDense2csr()**.

The routine **cusparse<t>pruneDense2csrNnz()** has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

The routine **cusparse<t>DpruneDense2csr()** has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.
A	device	array of dimension (lda, n).
lda	device	leading dimension of **A**. It must be at least max(1, m).
threshold	host or device	a value to drop the entries of A. **threshold** can point to a device memory or host memory.
descrC	host	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
pBuffer	device	buffer allocated by the user; the size is returned by **pruneDense2csr_bufferSizeExt()**.

Output

parameter	device or host	description
nnzTotalDevHostPtr	**device or host**	total number of nonzero of matrix `C`. **nnzTotalDevHostPtr** can point to a device memory or host memory.
csrValC	**device**	<type> array of **nnzC** nonzero elements of matrix `C`.
csrRowsPtrC	**device**	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	**device**	integer array of **nnzC** column indices of `C`.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

See `cusparseStatus_t` for the description of the return status

13.22. cusparseXpruneCsr2csr()

```
cusparseStatus_t
cusparseHpruneCsr2csr_bufferSizeExt(cusparseHandle_t         handle,
                                    int                      m,
                                    int                      n,
                                    int                      nnzA,
                                    const cusparseMatDescr_t descrA,
                                    const __half*            csrValA,
                                    const int*               csrRowPtrA,
                                    const int*               csrColIndA,
                                    const __half*            threshold,
                                    const cusparseMatDescr_t descrC,
                                    const __half*            csrValC,
                                    const int*               csrRowPtrC,
                                    const int*               csrColIndC,
                                    size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseSpruneCsr2csr_bufferSizeExt(cusparseHandle_t         handle,
                                    int                      m,
                                    int                      n,
                                    int                      nnzA,
                                    const cusparseMatDescr_t descrA,
                                    const float*             csrValA,
                                    const int*               csrRowPtrA,
                                    const int*               csrColIndA,
                                    const float*             threshold,
                                    const cusparseMatDescr_t descrC,
                                    const float*             csrValC,
                                    const int*               csrRowPtrC,
                                    const int*               csrColIndC,
                                    size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseDpruneCsr2csr_bufferSizeExt(cusparseHandle_t         handle,
                                    int                      m,
                                    int                      n,
                                    int                      nnzA,
                                    const cusparseMatDescr_t descrA,
                                    const double*            csrValA,
                                    const int*               csrRowPtrA,
                                    const int*               csrColIndA,
                                    const double*            threshold,
                                    const cusparseMatDescr_t descrC,
                                    const double*            csrValC,
                                    const int*               csrRowPtrC,
                                    const int*               csrColIndC,
                                    size_t*
 pBufferSizeInBytes)
```

```
cusparseStatus_t
cusparseHpruneCsr2csrNnz(cusparseHandle_t         handle,
                         int                      m,
                         int                      n,
                         int                      nnzA,
                         const cusparseMatDescr_t descrA,
                         const __half*            csrValA,
                         const int*               csrRowPtrA,
                         const int*               csrColIndA,
                         const __half*            threshold,
                         const cusparseMatDescr_t descrC,
```

This function prunes a sparse matrix to a sparse matrix with CSR format.

Given a sparse matrix **A** and a non-negative value **threshold**, the function returns a sparse matrix **C**, defined by

$$C(i,j) = A(i,j) \quad \text{if } |A(i,j)| > \text{threshold}$$

The implementation adopts a two-step approach to do the conversion. First, the user allocates **csrRowPtrC** of **m+1** elements and uses function **pruneCsr2csrNnz()** to determine the number of nonzeros columns per row. Second, the user gathers **nnzC** (number of nonzeros of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC[m]-csrRowPtrC[0])** and allocates **csrValC** of **nnzC** elements and **csrColIndC** of **nnzC** integers. Finally function **pruneCsr2csr()** is called to complete the conversion.

The user must obtain the size of the buffer required by **pruneCsr2csr()** by calling **pruneCsr2csr_bufferSizeExt()**, allocate the buffer, and pass the buffer pointer to **pruneCsr2csr()**.

Appendix section provides a simple example of **pruneCsr2csr()**.

The routine **cusparse<t>pruneCsr2csrNnz()** has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

The routine **cusparse<t>pruneCsr2csr()** has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.
nnzA	host	number of nonzeros of matrix **A**.
descrA	host	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	device	<type> array of **nnzA** nonzero elements of matrix **A**.
csrRowsPtrA	device	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	device	integer array of **nnzA** column indices of **A**.
threshold	host or device	a value to drop the entries of A. **threshold** can point to a device memory or host memory.

descrC	host	the descriptor of matrix C. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
pBuffer	device	buffer allocated by the user; the size is returned by **pruneCsr2csr_bufferSizeExt()**.

Output

parameter	device or host	description
nnzTotalDevHostPtr	device or host	total number of nonzero of matrix C. **nnzTotalDevHostPtr** can point to a device memory or host memory.
csrValC	device	<type> array of **nnzC** nonzero elements of matrix C.
csrRowsPtrC	device	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	device	integer array of **nnzC** column indices of C.
pBufferSizeInBytes	host	number of bytes of the buffer.

See `cusparseStatus_t` for the description of the return status

13.23. cusparseXpruneDense2csrPercentage()

```
cusparseStatus_t
cusparseHpruneDense2csrByPercentage_bufferSizeExt(cusparseHandle_t
   handle,
                                                  int
   m,
                                                  int
   n,
                                                  const __half*
   A,
                                                  int
   lda,
                                                  float
   percentage,
                                                  const
 cusparseMatDescr_t descrC,
                                                  const __half*
   csrValC,
                                                  const int*
   csrRowPtrC,
                                                  const int*
   csrColIndC,
                                                  pruneInfo_t
   info,
                                                  size_t*
   pBufferSizeInBytes)

cusparseStatus_t
cusparseSpruneDense2csrByPercentage_bufferSizeExt(cusparseHandle_t
   handle,
                                                  int
   m,
                                                  int
   n,
                                                  const float*
   A,
                                                  int
   lda,
                                                  float
   percentage,
                                                  const
 cusparseMatDescr_t descrC,
                                                  const float*
   csrValC,
                                                  const int*
   csrRowPtrC,
                                                  const int*
   csrColIndC,
                                                  pruneInfo_t
   info,
                                                  size_t*
   pBufferSizeInBytes)

cusparseStatus_t
cusparseDpruneDense2csrByPercentage_bufferSizeExt(cusparseHandle_t
   handle,
                                                  int
   m,
                                                  int
   n,
                                                  const double*
   A,
                                                  int
   lda,
```

This function prunes a dense matrix to a sparse matrix by percentage.

Given a dense matrix **A** and a non-negative value **percentage**, the function computes sparse matrix **C** by the following three steps:

Step 1: sort absolute value of **A** in ascending order.

$$\text{key} := \text{sort}(\ |A|\ )$$

Step 2: choose threshold by the parameter **percentage**

$$\begin{aligned}
\text{pos} &= \text{ceil}(m*n*(\text{percentage}/100)) - 1\\
\text{pos} &= \min(\text{pos}, m*n-1)\\
\text{pos} &= \max(\text{pos}, 0)\\
\text{threshold} &= \text{key}[\text{pos}]
\end{aligned}$$

Step 3: call **pruneDense2csr()** by with the parameter **threshold**.

The implementation adopts a two-step approach to do the conversion. First, the user allocates **csrRowPtrC** of **m+1** elements and uses function **pruneDense2csrNnzByPercentage()** to determine the number of nonzeros columns per row. Second, the user gathers **nnzC** (number of nonzeros of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC[m]-csrRowPtrC[0])** and allocates **csrValC** of **nnzC** elements and **csrColIndC** of **nnzC** integers. Finally function **pruneDense2csrByPercentage()** is called to complete the conversion.

The user must obtain the size of the buffer required by **pruneDense2csrByPercentage()** by calling **pruneDense2csrByPercentage_bufferSizeExt()**, allocate the buffer, and pass the buffer pointer to **pruneDense2csrByPercentage()**.

Remark 1: the value of **percentage** must be not greater than 100. Otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Remark 2: the zeros of **A** are not ignored. All entries are sorted, including zeros. This is different from **pruneCsr2csrByPercentage()**

Appendix section provides a simple example of **pruneDense2csrNnzByPercentage()**.

The routine **cusparse<t>pruneDense2csrNnzByPercentage()** has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

The routine **cusparse<t>pruneDense2csrByPercentage()** has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

parameter	device or host	description

handle	**host**	handle to the cuSPARSE library context.
m	**host**	number of rows of matrix **A**.
n	**host**	number of columns of matrix **A**.
A	**device**	array of dimension (lda, n).
lda	**device**	leading dimension of **A**. It must be at least max(1, m).
percentage	**host**	percentage <=100 and percentage >= 0
descrC	**host**	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
pBuffer	**device**	buffer allocated by the user; the size is returned by **pruneDense2csrByPercentage_bufferSizeExt()**.

Output

parameter	**device or host**	**description**
nnzTotalDevHostPtr	**device or host**	total number of nonzero of matrix **C**. **nnzTotalDevHostPtr** can point to a device memory or host memory.
csrValC	**device**	<type> array of **nnzC** nonzero elements of matrix **C**.
csrRowsPtrC	**device**	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	**device**	integer array of **nnzC** column indices of **C**.
pBufferSizeInBytes	**host**	number of bytes of the buffer.

See `cusparseStatus_t` for the description of the return status

13.24. cusparseXpruneCsr2csrByPercentage()

```
cusparseStatus_t
cusparseHpruneCsr2csrByPercentage_bufferSizeExt(cusparseHandle_t
 handle,
                                                 int
 m,
                                                 int
 n,
                                                 int
 nnzA,
                                                 const cusparseMatDescr_t
 descrA,
                                                 const __half*
 csrValA,
                                                 const int*
 csrRowPtrA,
                                                 const int*
 csrColIndA,
                                                 float
 percentage,
                                                 const cusparseMatDescr_t
 descrC,
                                                 const __half*
 csrValC,
                                                 const int*
 csrRowPtrC,
                                                 const int*
 csrColIndC,
                                                 pruneInfo_t
 info,
                                                 size_t*
 pBufferSizeInBytes)

cusparseStatus_t
cusparseSpruneCsr2csrByPercentage_bufferSizeExt(cusparseHandle_t
 handle,
                                                 int
 m,
                                                 int
 n,
                                                 int
 nnzA,
                                                 const cusparseMatDescr_t
 descrA,
                                                 const float*
 csrValA,
                                                 const int*
 csrRowPtrA,
                                                 const int*
 csrColIndA,
                                                 float
 percentage,
                                                 const cusparseMatDescr_t
 descrC,
                                                 const float*
 csrValC,
                                                 const int*
 csrRowPtrC,
                                                 const int*
 csrColIndC,
                                                 pruneInfo_t
 info,
                                                 size_t*
 pBufferSizeInBytes)
```

This function prunes a sparse matrix to a sparse matrix by percentage.

Given a sparse matrix **A** and a non-negative value **percentage**, the function computes sparse matrix **C** by the following three steps:

Step 1: sort absolute value of **A** in ascending order.

$$\text{key} := \text{sort}(\ |\text{csrValA}|\ )$$

Step 2: choose threshold by the parameter **percentage**

$$\begin{aligned}\text{pos} &= \text{ceil}(\text{nnzA}*(\text{percentage}/100)) - 1\\ \text{pos} &= \min(\text{pos}, \text{nnzA}-1)\\ \text{pos} &= \max(\text{pos}, 0)\\ \text{threshold} &= \text{key}[\text{pos}]\end{aligned}$$

Step 3: call **pruneCsr2csr()** by with the parameter **threshold**.

The implementation adopts a two-step approach to do the conversion.
First, the user allocates **csrRowPtrC** of **m+1** elements and uses function **pruneCsr2csrNnzByPercentage()** to determine the number of nonzeros columns per row. Second, the user gathers **nnzC** (number of nonzeros of matrix **C**) from either **(nnzC=*nnzTotalDevHostPtr)** or **(nnzC=csrRowPtrC[m]-csrRowPtrC[0])** and allocates **csrValC** of **nnzC** elements and **csrColIndC** of **nnzC** integers. Finally function **pruneCsr2csrByPercentage()** is called to complete the conversion.

The user must obtain the size of the buffer required by **pruneCsr2csrByPercentage()** by calling **pruneCsr2csrByPercentage_bufferSizeExt()**, allocate the buffer, and pass the buffer pointer to **pruneCsr2csrByPercentage()**.

Remark 1: the value of **percentage** must be not greater than 100. Otherwise, **CUSPARSE_STATUS_INVALID_VALUE** is returned.

Appendix section provides a simple example of **pruneCsr2csrByPercentage()**.

The routine **cusparse<t>pruneCsr2csrNnzByPercentage()** has the following properties:

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution
- The routine does ***not*** support CUDA graph capture

The routine **cusparse<t>pruneCsr2csrByPercentage()** has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports CUDA graph capture

Input

parameter	device or host	description
handle	host	handle to the cuSPARSE library context.
m	host	number of rows of matrix **A**.
n	host	number of columns of matrix **A**.

nnzA	host	number of nonzeros of matrix **A**.
descrA	host	the descriptor of matrix **A**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
csrValA	device	<type> array of **nnzA** nonzero elements of matrix **A**.
csrRowsPtrA	device	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndA	device	integer array of **nnzA** column indices of **A**.
percentage	host	percentage <=100 and percentage >= 0
descrC	host	the descriptor of matrix **C**. The supported matrix type is **CUSPARSE_MATRIX_TYPE_GENERAL**, Also, the supported index bases are **CUSPARSE_INDEX_BASE_ZERO** and **CUSPARSE_INDEX_BASE_ONE**.
pBuffer	device	buffer allocated by the user; the size is returned by **pruneCsr2csrByPercentage_bufferSizeExt()**.

Output

parameter	device or host	description
nnzTotalDevHostPtr	device or host	total number of nonzero of matrix **C**. **nnzTotalDevHostPtr** can point to a device memory or host memory.
csrValC	device	<type> array of **nnzC** nonzero elements of matrix **C**.
csrRowsPtrC	device	integer array of **m+1** elements that contains the start of every row and the end of the last row plus one.
csrColIndC	device	integer array of **nnzC** column indices of **C**.
pBufferSizeInBytes	host	number of bytes of the buffer.

See `cusparseStatus_t` for the description of the return status

13.25. cusparse<t>nnz_compress()

```
cusparseStatus_t
cusparseSnnz_compress(cusparseHandle_t         handle,
                      int                      m,
                      const cusparseMatDescr_t descr,
                      const float*             csrValA,
                      const int*               csrRowPtrA,
                      int*                     nnzPerRow,
                      int*                     nnzC,
                      float                    tol)

cusparseStatus_t
cusparseDnnz_compress(cusparseHandle_t         handle,
                      int                      m,
                      const cusparseMatDescr_t descr,
                      const double*            csrValA,
                      const int*               csrRowPtrA,
                      int*                     nnzPerRow,
                      int*                     nnzC,
                      double                   tol)

cusparseStatus_t
cusparseCnnz_compress(cusparseHandle_t         handle,
                      int                      m,
                      const cusparseMatDescr_t descr,
                      const cuComplex*         csrValA,
                      const int*               csrRowPtrA,
                      int*                     nnzPerRow,
                      int*                     nnzC,
                      cuComplex                tol)

cusparseStatus_t
cusparseZnnz_compress(cusparseHandle_t         handle,
                      int                      m,
                      const cusparseMatDescr_t descr,
                      const cuDoubleComplex*   csrValA,
                      const int*               csrRowPtrA,
                      int*                     nnzPerRow,
                      int*                     nnzC,
                      cuDoubleComplex          tol)
```

This function is the step one to convert from csr format to compressed csr format.

Given a sparse matrix A and a non-negative value threshold, the function returns nnzPerRow(the number of nonzeros columns per row) and nnzC(the total number of nonzeros) of a sparse matrix C, defined by

$$C(i,j) = A(i,j) \quad \text{if } |A(i,j)| > \text{threshold}$$

A key assumption for the cuComplex and cuDoubleComplex case is that this tolerance is given as the real part. For example tol = 1e-8 + 0*i and we extract cureal, that is the x component of this struct.

- This function requires temporary extra storage that is allocated internally
- The routine does ***not*** support asynchronous execution

- The routine does ***not*** support CUDA graph capture

Input

`handle`	handle to the cuSPARSE library context.
`m`	number of rows of matrix **`A`**.
`descrA`	the descriptor of matrix **`A`**. The supported matrix type is **`CUSPARSE_MATRIX_TYPE_GENERAL`**. Also, the supported index bases are **`CUSPARSE_INDEX_BASE_ZERO`** and **`CUSPARSE_INDEX_BASE_ONE`**.
`csrValA`	csr noncompressed values array
`csrRowPtrA`	the corresponding input noncompressed row pointer.
`tol`	non-negative tolerance to determine if a number less than or equal to it.

Output

`nnzPerRow`	this array contains the number of elements whose absolute values are greater than tol per row.
`nnzC`	host/device pointer of the total number of elements whose absolute values are greater than tol.

See `cusparseStatus_t` for the description of the return status

Chapter 14. CUSPARSE GENERIC API REFERENCE

The cuSPARSE Generic APIs allow computing the most common sparse linear algebra operations, such as sparse matrix-vector (SpMV) and sparse matrix-matrix multiplication (SpMM), in a flexible way. The new APIs have the following capabilities and features:

- Set matrix data layouts, number of batches, and storage formats (for example, CSR, COO, and so on)
- Set input/output/compute data types. This also allows mixed data-type computation
- Set types of sparse matrix indices
- Choose the algorithm for the computation
- Provide external device memory for internal operations
- Provide extensive consistency checks across input matrices and vectors for a given routine. This includes the validation of matrix sizes, data types, layout, allowed operations, etc.

14.1. Generic Types Reference

The cuSPARSE generic type references are described in this section.

14.1.1. cudaDataType_t

The section describes the types shared by multiple CUDA Libraries and defined in the header file **`library_types.h`**. The **`cudaDataType`** type is an enumerator to specify the data precision. It is used when the data reference does not carry the type itself (e.g. **`void*`**). For example, it is used in the routine **`cusparseSpMM()`**.

Value	Meaning	Data Type	Header
`CUDA_R_16F`	The data type is 16-bit IEEE-754 floating-point	**`__half`**	`cuda_fp16.h`
`CUDA_C_16F`	The data type is 16-bit complex IEEE-754 floating-point	**`__half2`**	`cuda_fp16.h`
`CUDA_R_16BF`	The data type is 16-bit bfloat floating-point	**`__nv_bfloat16`**	`cuda_bf16.h`

Value	Meaning	Data Type	Header
CUDA_C_16BF	The data type is 16-bit complex bfloat floating-point	**__nv_bfloat162**	cuda_bf16.h
CUDA_R_32F	The data type is 32-bit IEEE-754 floating-point	**float**	
CUDA_C_32F	The data type is 32-bit complex IEEE-754 floating-point	**cuComplex**	cuComplex.h
CUDA_R_64F	The data type is 64-bit IEEE-754 floating-point	**double**	
CUDA_C_64F	The data type is 64-bit complex IEEE-754 floating-point	**cuDoubleComplex**	cuComplex.h
CUDA_R_8I	The data type is 8-bit integer	**int8_t**	stdint.h
CUDA_R_32I	The data type is 32-bit integer	**int32_t**	stdint.h

IMPORTANT: The Generic API routines allow all data types reported in the respective section of the documentation only on GPU architectures with *native* support for them. If a specific GPU model does not provide *native* support for a given data type, the routine returns **CUSPARSE_STATUS_ARCH_MISMATCH** error.

Unsupported data types and Compute Capability (CC):

- **__half** on GPUs with **CC < 53** (e.g. Kepler)
- **__nv_bfloat16** on GPUs with **CC < 80** (e.g. Kepler, Maxwell, Pascal, Volta, Turing)

see https://developer.nvidia.com/cuda-gpus

14.1.2. cusparseFormat_t

This type indicates the format of the sparse matrix.

Value	Meaning
CUSPARSE_FORMAT_COO	The matrix is stored in Coordinate (COO) format organized in *Structure of Arrays (SoA)* layout
CUSPARSE_FORMAT_COO_AOS	The matrix is stored in Coordinate (COO) format organized in *Array of Structures (SoA)* layout
CUSPARSE_FORMAT_CSR	The matrix is stored in Compressed Sparse Row (CSR) format

14.1.3. cusparseOrder_t

This type indicates the memory layout of a dense matrix. Currently, only column-major layout is supported.

Value	Meaning
CUSPARSE_ORDER_ROW	The matrix is stored in row-major
CUSPARSE_ORDER_COL	The matrix is stored in column-major

14.1.4. cusparseIndexType_t

This type indicates the index type for rappresenting the sparse matrix indices.

Value	Meaning
CUSPARSE_INDEX_16U	16-bit unsigned integer [1, 65535]
CUSPARSE_INDEX_32I	32-bit signed integer [1, 2^31 - 1]
CUSPARSE_INDEX_64I	64-bit signed integer [1, 2^63 - 1]

14.2. Sparse Vector APIs

The cuSPARSE helper functions for sparse vector descriptor are described in this section.

14.2.1. cusparseCreateSpVec()

```
cusparseStatus_t
cusparseCreateSpVec(cusparseSpVecDescr_t* spVecDescr,
                    int64_t               size,
                    int64_t               nnz,
                    void*                 indices,
                    void*                 values,
                    cusparseIndexType_t   idxType,
                    cusparseIndexBase_t   idxBase,
                    cudaDataType          valueType)
```

This function initializes the sparse matrix descriptor **spVecDescr**.

Param.	Memory	In/out	Meaning
spVecDescr	HOST	OUT	Sparse vector descriptor
size	HOST	IN	Size of the sparse vector
nnz	HOST	IN	Number of non-zero entries of the sparse vector
indices	DEVICE	IN	Indices of the sparse vector. Array of size **nnz**
values	DEVICE	IN	Values of the sparse vector. Array of size **nnz**
idxType	HOST	IN	Enumerator specifying the data type of **indices**
idxBase	HOST	IN	Enumerator specifying the the base index of **indices**
valueType	HOST	IN	Enumerator specifying the datatype of **values**

See `cusparseStatus_t` for the description of the return status

14.2.2. cusparseDestroySpVec()

```
cusparseStatus_t
cusparseDestroySpVec(cusparseSpVecDescr_t spVecDescr)
```

This function releases the host memory allocated for the sparse vector descriptor **spVecDescr**.

Param.	Memory	In/out	Meaning
spVecDescr	HOST	IN	Sparse vector descriptor

See `cusparseStatus_t` for the description of the return status

14.2.3. cusparseSpVecGet()

```
cusparseStatus_t
cusparseSpVecGet(const cusparseSpVecDescr_t spVecDescr,
                 int64_t*                   size,
                 int64_t*                   nnz,
                 void**                     indices,
                 void**                     values,
                 cusparseIndexType_t*       idxType,
                 cusparseIndexBase_t*       idxBase,
                 cudaDataType*              valueType)
```

This function returns the fields of the sparse vector descriptor **spVecDescr**.

Param.	Memory	In/out	Meaning
spVecDescr	HOST	IN	Sparse vector descriptor
size	HOST	OUT	Size of the sparse vector
nnz	HOST	OUT	Number of non-zero entries of the sparse vector
indices	DEVICE	OUT	Indices of the sparse vector. Array of size **nnz**
values	DEVICE	OUT	Values of the sparse vector. Array of size **nnz**
idxType	HOST	OUT	Enumerator specifying the data type of **indices**
idxBase	HOST	OUT	Enumerator specifying the the base index of **indices**
valueType	HOST	OUT	Enumerator specifying the datatype of **values**

See `cusparseStatus_t` for the description of the return status

14.2.4. cusparseSpVecGetIndexBase()

```
cusparseStatus_t
cusparseSpVecGetIndexBase(const cusparseSpVecDescr_t spVecDescr,
                          cusparseIndexBase_t*       idxBase)
```

This function returns the **idxBase** field of the sparse vector descriptor **spVecDescr**.

Param.	Memory	In/out	Meaning
spVecDescr	HOST	IN	Sparse vector descriptor
idxBase	HOST	OUT	Enumerator specifying the the base index of **indices**

See `cusparseStatus_t` for the description of the return status

14.2.5. cusparseSpVecGetValues()

```
cusparseStatus_t
cusparseSpVecGetValues(const cusparseSpVecDescr_t spVecDescr,
                       void**                     values)
```

This function returns the **values** field of the sparse vector descriptor **spVecDescr**.

Param.	Memory	In/out	Meaning
spVecDescr	HOST	IN	Sparse vector descriptor
values	DEVICE	OUT	Values of the sparse vector. Array of size **nnz**

See `cusparseStatus_t` for the description of the return status

14.2.6. cusparseSpVecSetValues()

```
cusparseStatus_t
cusparseSpVecSetValues(cusparseSpVecDescr_t spVecDescr,
                       void*                values)
```

This function set the **values** field of the sparse vector descriptor **spVecDescr**.

Param.	Memory	In/out	Meaning
spVecDescr	HOST	IN	Sparse vector descriptor
values	DEVICE	IN	Values of the sparse vector. Array of size **nnz**

See `cusparseStatus_t` for the description of the return status

14.3. Sparse Matrix APIs

The cuSPARSE helper functions for sparse matrix descriptor are described in this section.

14.3.1. cusparseCreateCoo()

```
cusparseStatus_t
cusparseCreateCoo(cusparseSpMatDescr_t* spMatDescr,
                  int64_t               rows,
                  int64_t               cols,
                  int64_t               nnz,
                  void*                 cooRowInd,
                  void*                 cooColInd,
                  void*                 cooValues,
                  cusparseIndexType_t   cooIdxType,
                  cusparseIndexBase_t   idxBase,
                  cudaDataType          valueType)
```

This function initializes the sparse matrix descriptor **spMatDescr** in the COO format (Structure of Arrays layout).

Param.	Memory	In/out	Meaning
`spMatDescr`	HOST	OUT	Sparse matrix descriptor
`rows`	HOST	IN	Number of rows of the sparse matrix
`cols`	HOST	IN	Number of columns of the sparse matrix
`nnz`	HOST	IN	Number of non-zero entries of the sparse matrix
`cooRowInd`	DEVICE	IN	Row indices of the sparse matrix. Array of size `nnz`
`cooColInd`	DEVICE	IN	Column indices of the sparse matrix. Array of size `nnz`
`cooValues`	DEVICE	IN	Values of the sparse martix. Array of size `nnz`
`cooIdxType`	HOST	IN	Enumerator specifying the data type of `cooRowInd` and `cooColInd`
`idxBase`	HOST	IN	Enumerator specifying the base index of `cooRowInd` and `cooColInd`
`valueType`	HOST	IN	Enumerator specifying the datatype of `cooValues`

See `cusparseStatus_t` for the description of the return status

14.3.2. cusparseCreateCooAoS()

```
cusparseStatus_t
cusparseCreateCooAoS(cusparseSpMatDescr_t* spMatDescr,
                     int64_t               rows,
                     int64_t               cols,
                     int64_t               nnz,
                     void*                 cooInd,
                     void*                 cooValues,
                     cusparseIndexType_t   cooIdxType,
                     cusparseIndexBase_t   idxBase,
                     cudaDataType          valueType)
```

This function initializes the sparse matrix descriptor **`spMatDescr`** in the COO format (Array of Structures layout).

Param.	Memory	In/out	Meaning
`spMatDescr`	HOST	OUT	Sparse matrix descriptor
`rows`	HOST	IN	Number of rows of the sparse matrix
`cols`	HOST	IN	Number of columns of the sparse matrix
`nnz`	HOST	IN	Number of non-zero entries of the sparse matrix
`cooInd`	DEVICE	IN	<Row, Column> indices of the sparse matrix. Array of size `nnz`
`cooValues`	DEVICE	IN	Values of the sparse martix. Array of size `nnz`
`cooIdxType`	HOST	IN	Enumerator specifying the data type of `cooInd`
`idxBase`	HOST	IN	Enumerator specifying the base index of `cooInd`
`valueType`	HOST	IN	Enumerator specifying the datatype of `cooValues`

See `cusparseStatus_t` for the description of the return status

14.3.3. cusparseCreateCsr()

```
cusparseStatus_t
cusparseCreateCsr(cusparseSpMatDescr_t* spMatDescr,
                  int64_t               rows,
                  int64_t               cols,
                  int64_t               nnz,
                  void*                 csrRowOffsets,
                  void*                 csrColInd,
                  void*                 csrValues,
                  cusparseIndexType_t   csrRowOffsetsType,
                  cusparseIndexType_t   csrColIndType,
                  cusparseIndexBase_t   idxBase,
                  cudaDataType          valueType)
```

This function initializes the sparse matrix descriptor **`spMatDescr`** in the CSR format.

Param.	Memory	In/out	Meaning
`spMatDescr`	HOST	OUT	Sparse matrix descriptor
`rows`	HOST	IN	Number of rows of the sparse matrix
`cols`	HOST	IN	Number of columns of the sparse matrix
`nnz`	HOST	IN	Number of non-zero entries of the sparse matrix
`csrRowOffs`	DEVICE	IN	Row offsets of the sparse matrix. Array of size **`rows + 1`**
`csrColInd`	DEVICE	IN	Column indices of the sparse matrix. Array of size **`nnz`**
`csrValues`	DEVICE	IN	Values of the sparse martix. Array of size **`nnz`**
`csrRowOffs`	HOST	IN	Enumerator specifying the data type of **`csrRowOffsets`**
`csrColIndT`	HOST	IN	Enumerator specifying the data type of **`csrColInd`**
`idxBase`	HOST	IN	Enumerator specifying the base index of **`csrRowOffsets`** and **`csrColInd`**
`valueType`	HOST	IN	Enumerator specifying the datatype of **`csrValues`**

See `cusparseStatus_t` for the description of the return status

14.3.4. cusparseDestroySpMat()

```
cusparseStatus_t
cusparseDestroySpMat(cusparseSpMatDescr_t spMatDescr)
```

This function releases the host memory allocated for the sparse matrix descriptor **`spMatDescr`**.

Param.	Memory	In/out	Meaning
`spMatDescr`	HOST	IN	Sparse matrix descriptor

See `cusparseStatus_t` for the description of the return status

14.3.5. cusparseCooGet()

```
cusparseStatus_t
cusparseCooGet(const cusparseSpMatDescr_t spMatDescr,
               int64_t*                   rows,
               int64_t*                   cols,
               int64_t*                   nnz,
               void**                     cooRowInd,
               void**                     cooColInd,
               void**                     cooValues,
               cusparseIndexType_t*       idxType,
               cusparseIndexBase_t*       idxBase,
               cudaDataType*              valueType)
```

This function returns the fields of the sparse matrix descriptor **spMatDescr** stored in COO format (Array of Structures layout).

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
rows	HOST	OUT	Number of rows of the sparse matrix
cols	HOST	OUT	Number of columns of the sparse matrix
nnz	HOST	OUT	Number of non-zero entries of the sparse matrix
cooRowInd	DEVICE	OUT	Row indices of the sparse matrix. Array of size **nnz**
cooColInd	DEVICE	OUT	Column indices of the sparse matrix. Array of size **nnz**
cooValues	DEVICE	OUT	Values of the sparse martix. Array of size **nnz**
cooIdxType	HOST	OUT	Enumerator specifying the data type of **cooRowInd** and **cooColInd**
idxBase	HOST	OUT	Enumerator specifying the base index of **cooRowInd** and **cooColInd**
valueType	HOST	OUT	Enumerator specifying the datatype of **cooValues**

See `cusparseStatus_t` for the description of the return status

14.3.6. cusparseCooAosGet()

```
cusparseStatus_t
cusparseCooAoSGet(const cusparseSpMatDescr_t spMatDescr,
                  int64_t*                   rows,
                  int64_t*                   cols,
                  int64_t*                   nnz,
                  void**                     cooInd,
                  void**                     cooValues,
                  cusparseIndexType_t*       idxType,
                  cusparseIndexBase_t*       idxBase,
                  cudaDataType*              valueType)
```

This function returns the fields of the sparse matrix descriptor **spMatDescr** stored in COO format (Structure of Arrays layout).

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
rows	HOST	OUT	Number of rows of the sparse matrix
cols	HOST	OUT	Number of columns of the sparse matrix
nnz	HOST	OUT	Number of non-zero entries of the sparse matrix
cooInd	DEVICE	OUT	<Row, Column> indices of the sparse matrix. Array of size **nnz**
cooValues	DEVICE	OUT	Values of the sparse martix. Array of size **nnz**
cooIdxType	HOST	OUT	Enumerator specifying the data type of **cooInd**
idxBase	HOST	OUT	Enumerator specifying the base index of **cooInd**
valueType	HOST	OUT	Enumerator specifying the datatype of **cooValues**

See `cusparseStatus_t` for the description of the return status

14.3.7. cusparseCsrGet()

```
cusparseStatus_t CUSPARSEAPI
cusparseCsrGet(const cusparseSpMatDescr_t spMatDescr,
               int64_t*                   rows,
               int64_t*                   cols,
               int64_t*                   nnz,
               void**                     csrRowOffsets,
               void**                     csrColInd,
               void**                     csrValues,
               cusparseIndexType_t*       csrRowOffsetsType,
               cusparseIndexType_t*       csrColIndType,
               cusparseIndexBase_t*       idxBase,
               cudaDataType*              valueType);
```

This function returns the fields of the sparse matrix descriptor **spMatDescr** stored in CSR format.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
rows	HOST	OUT	Number of rows of the sparse matrix
cols	HOST	OUT	Number of columns of the sparse matrix
nnz	HOST	OUT	Number of non-zero entries of the sparse matrix
csrRowOffs	DEVICE	OUT	Row offsets of the sparse matrix. Array of size **rows + 1**
csrColInd	DEVICE	OUT	Column indices of the sparse matrix. Array of size **nnz**
csrValues	DEVICE	OUT	Values of the sparse martix. Array of size **nnz**
csrRowOffs	HOST	OUT	Enumerator specifying the data type of **csrRowOffsets**
csrColIndT	HOST	OUT	Enumerator specifying the data type of **csrColInd**
idxBase	HOST	OUT	Enumerator specifying the base index of **csrRowOffsets** and **csrColInd**

Param.	Memory	In/out	Meaning
valueType	HOST	OUT	Enumerator specifying the datatype of **csrValues**

See cusparseStatus_t for the description of the return status

14.3.8. cusparseCsrSetPointers()

```
cusparseStatus_t CUSPARSEAPI
cusparseCsrSetPointers(cusparseSpMatDescr_t spMatDescr,
                       void*                csrRowOffsets,
                       void*                csrColInd,
                       void*                csrValues);
```

This function sets the pointers of the sparse matrix descriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
csrRowOffs	DEVICE	IN	Row offsets of the sparse martix. Array of size **rows + 1**
csrColInd	DEVICE	IN	Column indices of the sparse martix. Array of size **nnz**
csrValues	DEVICE	IN	Values of the sparse martix. Array of size **nnz**

See cusparseStatus_t for the description of the return status

14.3.9. cusparseSpMatGetSize()

```
cusparseStatus_t CUSPARSEAPI
cusparseSpMatGetSize(cusparseSpMatDescr_t spMatDescr,
                     int64_t*             rows,
                     int64_t*             cols,
                     int64_t*             nnz);
```

This function returns the sizes of the sparse matrix **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
rows	HOST	OUT	Number of rows of the sparse matrix
cols	HOST	OUT	Number of columns of the sparse matrix
nnz	HOST	OUT	Number of non-zero entries of the sparse matrix

See cusparseStatus_t for the description of the return status

14.3.10. cusparseSpMatGetFormat()

```
cusparseStatus_t
cusparseSpMatGetFormat(const cusparseSpMatDescr_t spMatDescr,
                       cusparseFormat_t*          format)
```

This function returns the **format** field of the sparse matrix descriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
format	HOST	OUT	Enumerator specifying the storage format of the sparse matrix

See `cusparseStatus_t` for the description of the return status

14.3.11. cusparseSpMatGetIndexBase()

```
cusparseStatus_t
cusparseSpMatGetIndexBase(const cusparseSpMatDescr_t spMatDescr,
                          cusparseIndexBase_t*       idxBase)
```

This function returns the **idxBase** field of the sparse matrix descriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
idxBase	HOST	OUT	Enumerator specifying the base index of the sparse matrix

See `cusparseStatus_t` for the description of the return status

14.3.12. cusparseSpMatGetValues()

```
cusparseStatus_t CUSPARSEAPI
cusparseSpMatGetValues(cusparseSpMatDescr_t spMatDescr,
                       void**               values)
```

This function returns the **values** field of the sparse matrix descriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
values	DEVICE	OUT	Values of the sparse martix. Array of size **nnz**

See `cusparseStatus_t` for the description of the return status

14.3.13. cusparseSpMatSetValues()

```
cusparseStatus_t CUSPARSEAPI
cusparseSpMatSetValues(cusparseSpMatDescr_t spMatDescr,
                       void*                values)
```

This function sets the **values** field of the sparse matrix descriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
values	DEVICE	IN	Values of the sparse martix. Array of size **nnz**

See `cusparseStatus_t` for the description of the return status

14.3.14. cusparseSpMatGetStridedBatch()

```
cusparseStatus_t
cusparseSpMatGetStridedBatch(const cusparseSpMatDescr_t spMatDescr,
                             int*                       batchCount)
```

This function returns the **batchCount** field of the sparse matrixdescriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
batchCount	HOST	OUT	Number of batches of the sparse matrix

See cusparseStatus_t for the description of the return status

14.3.15. cusparseSpMatSetStridedBatch() [DEPRECATED]

[[DEPRECATED]] use **cusparseSpMatSetCsrStridedBatch()** , **cusparseSpMatSetCooStridedBatch()** instead. *The routine will be removed in the next major release*

```
cusparseStatus_t
cusparseSpMatSetStridedBatch(cusparseSpMatDescr_t spMatDescr,
                             int                  batchCount)
```

This function sets the **batchCount** field of the sparse matrix descriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
batchCount	HOST	IN	Number of batches of the sparse matrix

See cusparseStatus_t for the description of the return status

14.3.16. cusparseCooSetStridedBatch()

```
cusparseStatus_t
cusparseCooSetStridedBatch(cusparseSpMatDescr_t spMatDescr,
                           int                  batchCount,
                           int64_t              batchStride)
```

This function sets the **batchCount** and the **batchStride** fields of the sparse matrix descriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
batchCount	HOST	IN	Number of batches of the sparse matrix
batchStrid	HOST	IN	address offset between consecutive batches

See cusparseStatus_t for the description of the return status

14.3.17. cusparseCsrSetStridedBatch()

```
cusparseStatus_t
cusparseCsrSetStridedBatch(cusparseSpMatDescr_t spMatDescr,
                           int                  batchCount,
                           int64_t              offsetsBatchStride,
                           int64_t
 columnsValuesBatchStride)
```

This function sets the **batchCount** and the **batchStride** fields of the sparse matrix descriptor **spMatDescr**.

Param.	Memory	In/out	Meaning
spMatDescr	HOST	IN	Sparse matrix descriptor
batchCount	HOST	IN	Number of batches of the sparse matrix
offsetsBat	HOST	IN	Address offset between consecutive batches for the row offset array
offsetsBat	HOST	IN	Address offset between consecutive batches for the column and value arrays

See `cusparseStatus_t` for the description of the return status

14.4. Dense Vector APIs

The cuSPARSE helper functions for dense vector descriptor are described in this section.

14.4.1. cusparseCreateDnVec()

```
cusparseStatus_t
cusparseCreateDnVec(cusparseDnVecDescr_t* dnVecDescr,
                    int64_t               size,
                    void*                 values,
                    cudaDataType          valueType)
```

This function initializes the dense vector descriptor **dnVecDescr**.

Param.	Memory	In/out	Meaning
dnVecDescr	HOST	OUT	Dense vector descriptor
size	HOST	IN	Size of the dense vector
values	DEVICE	IN	Values of the dense vector. Array of size **size**
valueType	HOST	IN	Enumerator specifying the datatype of **values**

See `cusparseStatus_t` for the description of the return status

14.4.2. cusparseDestroyDnVec()

```
cusparseStatus_t
cusparseDestroyDnVec(cusparseDnVecDescr_t dnVecDescr)
```

This function releases the host memory allocated for the dense vector descriptor **dnVecDescr**.

Param.	Memory	In/out	Meaning
dnVecDescr	HOST	IN	Dense vector descriptor

See cusparseStatus_t for the description of the return status

14.4.3. cusparseDnVecGet()

```
cusparseStatus_t
cusparseDnVecGet(const cusparseDnVecDescr_t dnVecDescr,
                 int64_t*                   size,
                 void**                     values,
                 cudaDataType*              valueType)
```

This function returns the fields of the dense vector descriptor **dnVecDescr**.

Param.	Memory	In/out	Meaning
dnVecDescr	HOST	IN	Dense vector descriptor
size	HOST	OUT	Size of the dense vector
values	DEVICE	OUT	Values of the dense vector. Array of size **nnz**
valueType	HOST	OUT	Enumerator specifying the datatype of **values**

See cusparseStatus_t for the description of the return status

14.4.4. cusparseDnVecGetValues()

```
cusparseStatus_t
cusparseDnVecGetValues(const cusparseDnVecDescr_t dnVecDescr,
                       void**                     values)
```

This function returns the **values** field of the dense vector descriptor **dnVecDescr**.

Param.	Memory	In/out	Meaning
dnVecDescr	HOST	IN	Dense vector descriptor
values	DEVICE	OUT	Values of the dense vector

See cusparseStatus_t for the description of the return status

14.4.5. cusparseDnVecSetValues()

```
cusparseStatus_t
cusparseDnVecSetValues(cusparseDnVecDescr_t dnVecDescr,
                       void*                values)
```

This function set the **values** field of the dense vector descriptor **dnVecDescr**.

Param.	Memory	In/out	Meaning
dnVecDescr	HOST	IN	Dense vector descriptor
values	DEVICE	IN	Values of the dense vector. Array of size **size**

The possible error values returned by this function and their meanings are listed below :

See cusparseStatus_t for the description of the return status

14.5. Dense Matrix APIs

The cuSPARSE helper functions for dense matrix descriptor are described in this section.

14.5.1. cusparseCreateDnMat()

```
cusparseStatus_t
cusparseCreateDnMat(cusparseDnMatDescr_t* dnMatDescr,
                    int64_t               rows,
                    int64_t               cols,
                    int64_t               ld,
                    void*                 values,
                    cudaDataType          valueType,
                    cusparseOrder_t       order)
```

The function initializes the dense matrix descriptor **dnMatDescr**.

Param.	Memory	In/out	Meaning
dnMatDescr	HOST	OUT	Dense matrix descriptor
rows	HOST	IN	Number of rows of the dense matrix
cols	HOST	IN	Number of columns of the dense matrix
ld	HOST	IN	Leading dimension of the dense matrix
values	DEVICE	IN	Values of the dense matrix. Array of size **size**
valueType	HOST	IN	Enumerator specifying the datatype of **values**
order	HOST	IN	Enumerator specifying the memory layout of the dense matrix

See cusparseStatus_t for the description of the return status

14.5.2. cusparseDestroyDnMat()

```
cusparseStatus_t
cusparseDestroyDnMat(cusparseDnMatDescr_t dnMatDescr)
```

This function releases the host memory allocated for the dense matrix descriptor **dnMatDescr**.

Param.	Memory	In/out	Meaning
dnMatDescr	HOST	IN	Dense matrix descriptor

See cusparseStatus_t for the description of the return status

14.5.3. cusparseDnMatGet()

```
cusparseStatus_t
cusparseDnMatGet(const cusparseDnMatDescr_t dnMatDescr,
                 int64_t*                   rows,
                 int64_t*                   cols,
                 int64_t*                   ld,
                 void**                     values,
                 cudaDataType*              type,
                 cusparseOrder_t*           order)
```

This function returns the fields of the dense matrix descriptor **dnMatDescr**.

Param.	Memory	In/out	Meaning
dnMatDescr	HOST	IN	Dense matrix descriptor
rows	HOST	OUT	Number of rows of the dense matrix
cols	HOST	OUT	Number of columns of the dense matrix
ld	HOST	OUT	Leading dimension of the dense matrix
values	DEVICE	OUT	Values of the dense matrix. Array of size **ld * cols**
valueType	HOST	OUT	Enumerator specifying the datatype of **values**
order	HOST	OUT	Enumerator specifying the memory layout of the dense matrix

See cusparseStatus_t for the description of the return status

14.5.4. cusparseDnMatGetValues()

```
cusparseStatus_t CUSPARSEAPI
cusparseDnMatGetValues(const cusparseDnMatDescr_t dnMatDescr,
                       void**                     values)
```

This function returns the **values** field of the dense matrix descriptor **dnMatDescr**.

Param.	Memory	In/out	Meaning
dnMatDescr	HOST	IN	Dense matrix descriptor

Param.	Memory	In/out	Meaning
values	DEVICE	OUT	Values of the dense matrix. Array of size **ld * cols**

See cusparseStatus_t for the description of the return status

14.5.5. cusparseDnSetValues()

```
cusparseStatus_t CUSPARSEAPI
cusparseDnMatSetValues(cusparseDnMatDescr_t dnMatDescr,
                       void*                values)
```

This function sets the **values** field of the dense matrix descriptor **dnMatDescr**.

Param.	Memory	In/out	Meaning
dnMatDescr	HOST	IN	Dense matrix descriptor
values	DEVICE	IN	Values of the dense matrix. Array of size **ld * cols**

See cusparseStatus_t for the description of the return status

14.5.6. cusparseDnMatGetStridedBatch()

```
cusparseStatus_t
cusparseDnMatGetStridedBatch(const cusparseDnMatDescr_t dnMatDescr,
                             int*                       batchCount,
                             int64_t*                   batchStride)
```

The function returns the number of batches and the batch stride of the dense matrix descriptor **dnMatDescr**.

Param.	Memory	In/out	Meaning
dnMatDescr	HOST	IN	Dense matrix descriptor
batchCount	HOST	OUT	Number of batches of the dense matrix
batchStrid	HOST	OUT	Address offset between a matrix and the next one in the batch

See cusparseStatus_t for the description of the return status

14.5.7. cusparseDnMatSetStridedBatch()

```
cusparseStatus_t
cusparseDnMatSetStridedBatch(cusparseDnMatDescr_t dnMatDescr,
                             int                  batchCount,
                             int64_t              batchStride)
```

The function sets the number of batches and the batch stride of the dense matrix descriptor **dnMatDescr**.

Param.	Memory	In/out	Meaning
dnMatDescr	HOST	IN	Dense matrix descriptor

Param.	Memory	In/out	Meaning
batchCount	HOST	IN	Number of batches of the dense matrix
batchStrid	HOST	IN	Address offset between a matrix and the next one in the batch. **batchStride** ≥ **ld * cols** if the matrix uses column-major layout, **batchStride** ≥ **ld * rows** otherwise

See cusparseStatus_t for the description of the return status

14.6. Generic API Functions

14.6.1. cusparseAxpby()

```
cusparseStatus_t
cusparseAxpby(cusparseHandle_t     handle,
              const void*          alpha,
              cusparseSpVecDescr_t vecX,
              const void*          beta,
              cusparseDnVecDescr_t vecY)
```

The function computes the sum of a sparse vector **vecX** and a dense vector **vecY**

$$\mathbf{Y} = \alpha\mathbf{X} + \beta\mathbf{Y}$$

In other words,

```
for i=0 to nnz-1
    Y[X_indices[i]] = alpha * X_values[i] + beta * Y[X_indices[i]]
```

Param.	Memory	In/out	Meaning
handle	HOST	IN	Handle to the cuSPARSE library context
alpha	HOST or DEVICE	IN	α scalar used for multiplication
vecX	HOST	IN	Sparse vector **X**
beta	HOST or DEVICE	IN	β scalar used for multiplication
vecY	HOST	IN/OUT	Dense vector **Y**

cusparseAxpby supports the following index type for representing the sparse vector **vecX**:

- 32-bit indices (**CUSPARSE_INDEX_32I**)
- 64-bit indices (**CUSPARSE_INDEX_64I**)

cusparseAxpby supports the following datatypes:

X/Y
CUDA_R_16F

X/Y
`CUDA_R_16BF`
`CUDA_R_32F`
`CUDA_R_64F`
`CUDA_C_16F`
`CUDA_C_16BF`
`CUDA_C_32F`
`CUDA_C_64F`

`cusparseAxpby()` has the following constraints:

- The arrays representing the sparse vector **`vecX`** must be aligned to 16 bytes

`cusparseAxpby()` has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector **`vecX`** indices are distinct

`cusparseAxpby()` supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseAxpby for a code example.

14.6.2. cusparseGather()

```
cusparseStatus_t
cusparseGather(cusparseHandle_t     handle,
               cusparseDnVecDescr_t vecY,
               cusparseSpVecDescr_t vecX)
```

The function gathers the elements of the dense vector **`vecY`** into the sparse vector **`vecX`**

In other words,

```
for i=0 to nnz-1
    X_values[i] = Y[X_indices[i]]
```

Param.	Memory	In/out	Meaning
`handle`	HOST	IN	Handle to the cuSPARSE library context
`vecX`	HOST	OUT	Sparse vector **`X`**
`vecY`	HOST	IN	Dense vector **`Y`**

cusparseGather supports the following index type for representing the sparse vector **vecX**:

- 32-bit indices (**CUSPARSE_INDEX_32I**)
- 64-bit indices (**CUSPARSE_INDEX_64I**)

cusparseGather supports the following datatypes:

X/Y
CUDA_R_16F
CUDA_R_16BF
CUDA_R_32F
CUDA_R_64F
CUDA_C_16F
CUDA_C_16BF
CUDA_C_32F
CUDA_C_64F

cusparseGather() has the following constraints:

- The arrays representing the sparse vector **vecX** must be aligned to 16 bytes

cusparseGather() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector **vecX** indices are distinct

cusparseGather() supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseGather for a code example.

14.6.3. cusparseScatter()

```
cusparseStatus_t
cusparseScatter(cusparseHandle_t     handle,
                cusparseSpVecDescr_t vecX,
                cusparseDnVecDescr_t vecY)
```

The function scatters the elements of the sparse vector **vecX** into the dense vector **vecY**

In other words,

```
for i=0 to nnz-1
    Y[X_indices[i]] = X_values[i]
```

Param.	Memory	In/out	Meaning
handle	HOST	IN	Handle to the cuSPARSE library context
vecX	HOST	IN	Sparse vector **X**
vecY	HOST	OUT	Dense vector **Y**

cusparseScatter supports the following index type for representing the sparse vector **vecX**:

- 32-bit indices (**CUSPARSE_INDEX_32I**)
- 64-bit indices (**CUSPARSE_INDEX_64I**)

cusparseScatter supports the following datatypes:

X/Y
CUDA_R_16F
CUDA_R_16BF
CUDA_R_32F
CUDA_R_64F
CUDA_C_16F
CUDA_C_16BF
CUDA_C_32F
CUDA_C_64F

cusparseScatter() has the following constraints:

- The arrays representing the sparse vector **vecX** must be aligned to 16 bytes

cusparseScatter() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector **vecX** indices are distinct

cusparseScatter() supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseScatter for a code example.

14.6.4. cusparseRot()

```
cusparseStatus_t
cusparseRot(cusparseHandle_t     handle,
            const void*          c_coeff,
            const void*          s_coeff,
            cusparseSpVecDescr_t vecX,
            cusparseDnVecDescr_t vecY)
```

The function computes the Givens rotation matrix

$$G = \begin{bmatrix} c & s \\ -s & c \end{bmatrix}$$

to a sparse **vecX** and a dense vector **vecY**

In other words,

```
for i=0 to nnz-1
    Y[X_indices[i]] = c * Y[X_indices[i]] - s * X_values[i]
    X_values[i]     = c * X_values[i]     + s * Y[X_indices[i]]
```

Param.	Memory	In/out	Meaning
handle	HOST	IN	Handle to the cuSPARSE library context
c_coeff	HOST or DEVICE	IN	cosine element of the rotation matrix
vecX	HOST	IN/OUT	Sparse vector **X**
s_coeff	HOST or DEVICE	IN	sine element of the rotation matrix
vecY	HOST	IN/OUT	Dense vector **Y**

cusparseRot supports the following index type for representing the sparse vector **vecX**:

- 32-bit indices (**CUSPARSE_INDEX_32I**)
- 64-bit indices (**CUSPARSE_INDEX_64I**)

cusparseRot supports the following datatypes:

X/Y
CUDA_R_16F
CUDA_R_16BF
CUDA_R_32F
CUDA_R_64F
CUDA_C_16F
CUDA_C_16BF
CUDA_C_32F
CUDA_C_64F

cusparseRot() has the following constraints:

- The arrays representing the sparse vector **vecX** must be aligned to 16 bytes

cusparseRot() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector **vecX** indices are distinct

cusparseRot() supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseRot for a code example.

14.6.5. cusparseSpVV()

```
cusparseStatus_t
cusparseSpVV_bufferSize(cusparseHandle_t     handle,
                        cusparseOperation_t  opX,
                        cusparseSpVecDescr_t vecX,
                        cusparseDnVecDescr_t vecY,
                        void*                result,
                        cudaDataType         computeType,
                        size_t*              bufferSize)
```

```
cusparseStatus_t
cusparseSpVV(cusparseHandle_t     handle,
             cusparseOperation_t  opX,
             cusparseSpVecDescr_t vecX,
             cusparseDnVecDescr_t vecY,
             void*                result,
             cudaDataType         computeType,
             void*                externalBuffer)
```

The function computes the inner dot product of a sparse vector **vecX** and a dense vector **vecY**

$$result = \mathbf{X'} \cdot \mathbf{Y}$$

In other words,

```
result = 0;
for i=0 to nnz-1
    result += X_values[i] * Y[X_indices[i]]
```

$$\text{op}(X) = \begin{cases} X & \text{if op(X) == CUSPARSE\_OPERATION\_ NON\_TRANSPOSE} \\ \overline{X} & \text{if op(X) == CUSPARSE\_OPERATION\_ CONJUGATE\_TRANSPOSE} \end{cases}$$

The function **`cusparseSpVV_bufferSize()`** returns the size of the workspace needed by **`cusparseSpVV()`**

Param.	Memory	In/out	Meaning
`handle`	HOST	IN	Handle to the cuSPARSE library context
`opX`	HOST	IN	Operation `op(X)` that is non-transpose or conjugate transpose
`vecX`	HOST	IN	Sparse vector `X`
`vecY`	HOST	IN	Dense vector `Y`
`result`	HOST or DEVICE	OUT	The resulting dot product
`computeType`	HOST	IN	Enumerator specifying the datatype in which the computation is executed
`bufferSize`	HOST	OUT	Number of bytes of workspace needed by `cusparseSpVV`
`externalBuf`	DEVICE	IN	Pointer to workspace buffer

`cusparseSpVV` supports the follwing index type for representing the sparse vector **`vecX`**:

- 32-bit indices (**`CUSPARSE_INDEX_32I`**)
- 64-bit indices (**`CUSPARSE_INDEX_64I`**)

The datatypes combinations currrently supported for **`cusparseSpVV`** are listed below:

Uniform-precision computation:

X/Y/computeType
`CUDA_R_16F`
`CUDA_R_16BF`
`CUDA_R_32F`
`CUDA_R_64F`
`CUDA_C_16F`
`CUDA_C_16BF`
`CUDA_C_32F`
`CUDA_C_64F`

Mixed-precison computation:

X/Y	computeType/result
`CUDA_R_8I`	`CUDA_R_32I`
`CUDA_R_8I`	`CUDA_R_32F`
`CUDA_R_16F`	
`CUDA_R_16BF`	

`cusparseSpVV()` has the following constraints:

- The arrays representing the sparse vector **vecX** must be aligned to 16 bytes

cusparseSpVV() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run if the the sparse vector **vecX** indices are distinct

cusparseSpVV() supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseSpVV for a code example.

14.6.6. cusparseSpMV()

```
cusparseStatus_t
cusparseSpMV_bufferSize(cusparseHandle_t           handle,
                        cusparseOperation_t        opA,
                        const void*                alpha,
                        const cusparseSpMatDescr_t matA,
                        const cusparseDnVecDescr_t vecX,
                        const void*                beta,
                        const cusparseDnVecDescr_t vecY,
                        cudaDataType               computeType,
                        cusparseSpMVAlg_t          alg,
                        size_t*                    bufferSize)
```

```
cusparseStatus_t
cusparseSpMV(cusparseHandle_t           handle,
             cusparseOperation_t        opA,
             const void*                alpha,
             const cusparseSpMatDescr_t matA,
             const cusparseDnVecDescr_t vecX,
             const void*                beta,
             const cusparseDnVecDescr_t vecY,
             cudaDataType               computeType,
             cusparseSpMVAlg_t          alg,
             void*                      externalBuffer)
```

This function performs the multiplication of a sparse matrix **matA** and a dense vector **vecX**

$$\mathbf{Y} = \alpha op(\mathbf{A}) \cdot \mathbf{X} + \beta \mathbf{Y}$$

where

- **op(A)** is a sparse matrix of size $m \times k$
- **X** is a dense vector of size k
- **Y** is a dense vector of size m

- α and β are scalars

Also, for matrix **A**

$$\text{op}(A) = \begin{cases} A & \text{if op(A) == CUSPARSE\_ OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if op(A) == CUSPARSE\_ OPERATION\_TRANSPOSE} \\ A^H & \text{if op(A) == CUSPARSE\_ OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

When using the (conjugate) transpose of the sparse matrix **A**, this routine may produce slightly different results during different runs with the same input parameters.

The function **cusparseSpMV_bufferSize()** returns the size of the workspace needed by **cusparseSpMV()**

Param.	Memory	In/out	Meaning
handle	HOST	IN	Handle to the cuSPARSE library context
opA	HOST	IN	Operation **op(A)**
alpha	HOST or DEVICE	IN	α scalar used for multiplication
matA	HOST	IN	Sparse matrix **A**
vecX	HOST	IN	Dense vector **X**
beta	HOST or DEVICE	IN	β scalar used for multiplication
vecY	HOST	IN/OUT	Dense vector **Y**
computeTyp	HOST	IN	Enumerator specifying the datatype in which the computation is executed
alg	HOST	IN	Enumerator specifying the algorithm for the computation
bufferSize	HOST	OUT	Number of bytes of workspace needed by **cusparseSpMV**
externalBu	DEVICE	IN	Pointer to workspace buffer

cusparseSpMV supports the follwing index type for representing the sparse vector **vecX**:

- 32-bit indices (**CUSPARSE_INDEX_32I**)
- 64-bit indices (**CUSPARSE_INDEX_64I**)

cusparseSpMV supports the following datatypes:

Uniform-precision computation:

A/X/ Y/computeType
CUDA_R_16F
CUDA_R_32F
CUDA_R_64F
CUDA_C_16F
CUDA_C_32F
CUDA_C_64F

Mixed-precision computation:

A/X	Y	computeType
CUDA_R_8I	CUDA_R_32I	CUDA_R_32I
CUDA_R_8I	CUDA_R_32F	CUDA_R_32F
CUDA_R_16F		
CUDA_R_16F	CUDA_R_16F	

The sparse matrix formats currrently supported are listed below :

Format	Notes
CUSPARSE_FORMAT_COO	May produce slightly different results during different runs with the same input parameters
CUSPARSE_FORMAT_COO_	May produce slightly different results during different runs with the same input parameters
CUSPARSE_FORMAT_CSR	Provides deterministic (bit-wise) results for each run

cusparseSpMV supports the following algorithms:

Algorithm	Notes
CUSPARSE_MV_ALG_DEFA	Default algorithm for any sparse matrix format
CUSPARSE_COOMV_ALG	Default algorithm for COO sparse matrix format
CUSPARSE_CSRMV_ALG1	Default algorithm for CSR sparse matrix format
CUSPARSE_CSRMV_ALG2	Algorithm 2 for CSR sparse matrix format. May provide better performance for irregular matrices

The function has the following limitations:

- Half-precision is not supported with 64-bit indices (**CUSPARSE_INDEX_64I**)

cusparseSpMV() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution

cusparseSpMV() supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseSpMV CSR and cusparseSpMV COO for a code example.

14.6.7. cusparseSpMM()

```
cusparseStatus_t
cusparseSpMM_bufferSize(cusparseHandle_t     handle,
                        cusparseOperation_t  opA,
                        cusparseOperation_t  opB,
                        const void*          alpha,
                        cusparseSpMatDescr_t matA,
                        cusparseDnMatDescr_t matB,
                        const void*          beta,
                        cusparseDnMatDescr_t matC,
                        cudaDataType         computeType,
                        cusparseSpMMAlg_t    alg,
                        size_t*              bufferSize)
```

```
cusparseStatus_t
cusparseSpMM(cusparseHandle_t     handle,
             cusparseOperation_t  opA,
             cusparseOperation_t  opB,
             const void*          alpha,
             cusparseSpMatDescr_t matA,
             cusparseDnMatDescr_t matB,
             const void*          beta,
             cusparseDnMatDescr_t matC,
             cudaDataType         computeType,
             cusparseSpMMAlg_t    alg,
             void*                externalBuffer)
```

The function performs the multiplication of a sparse matrix **matA** and a dense matrix **matB**

$$\mathbf{C} = \alpha op(\mathbf{A}) \cdot op(\mathbf{B}) + \beta \mathbf{C}$$

where

- **op(A)** is a sparse matrix of size $m \times k$
- **op(B)** is a dense matrix of size $k \times n$
- **C** is a dense matrix of size $m \times n$
- α and β are scalars

The routine can be also used to perform the multiplication of a dense matrix **matB** and a sparse matrix **matA** by switching the dense matrices layout:

$$\mathbf{C}_C = \mathbf{B}_C \cdot \mathbf{A} + \beta \mathbf{C}_C \longrightarrow$$
$$\mathbf{C}_R = \mathbf{A}^T \cdot \mathbf{B}_R + \beta \mathbf{C}_R$$

where $\mathbf{B}_C$, $\mathbf{C}_C$ indicate column-major layout, while $\mathbf{B}_R$, $\mathbf{C}_R$ refer to row-major layout

Also, for matrix **A** and **B**

$$\text{op}(A) = \begin{cases} A & \text{if op(A) == CUSPARSE\_ OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if op(A) == CUSPARSE\_ OPERATION\_TRANSPOSE} \\ A^H & \text{if op(A) == CUSPARSE\_ OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

$$op(B) = \begin{cases} B & \text{if op(B) == CUSPARSE\_ OPERATION\_NON\_TRANSPOSE} \\ B^T & \text{if op(B) == CUSPARSE\_ OPERATION\_TRANSPOSE} \\ B^H & \text{if op(B) == CUSPARSE\_ OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

When using the (conjugate) transpose of the sparse matrix **A**, this routine may produce slightly different results during different runs with the same input parameters.

The function **cusparseSpMM_bufferSize()** returns the size of the workspace needed by **cusparseSpMM()**

Param.	Memory	In/out	Meaning
handle	HOST	IN	Handle to the cuSPARSE library context
opA	HOST	IN	Operation **op(A)**
alpha	HOST or DEVICE	IN	α scalar used for multiplication
matA	HOST	IN	Sparse matrix **A**
matB	HOST	IN	Dense matrix **B**
beta	HOST or DEVICE	IN	β scalar used for multiplication
matC	HOST	IN/OUT	Dense matrix **C**
computeTyp	HOST	IN	Enumerator specifying the datatype in which the computation is executed
alg	HOST	IN	Enumerator specifying the algorithm for the computation
bufferSize	HOST	OUT	Number of bytes of workspace needed by **cusparseSpMM**
externalBu	DEVICE	IN	Pointer to workspace buffer

cusparseSpMM supports the following sparse matrix formats:

- **CUSPARSE_FORMAT_COO**
- **CUSPARSE_FORMAT_CSR**

cusparseSpMM supports the following index type for representing the sparse vector **vecX**:

- 32-bit indices (**CUSPARSE_INDEX_32I**)
- 64-bit indices (**CUSPARSE_INDEX_64I**) only with **CUSPARSE_SPMM_COO_ALG4** and **CUSPARSE_SPMM_CSR_ALG2** algorithms

cusparseSpMM supports the following datatypes:

Uniform-precision computation:

A/B/ C/computeType
CUDA_R_16F
CUDA_R_16BF
CUDA_R_32F
CUDA_R_64F

A/B/ C/computeType
CUDA_C_16F
CUDA_C_16BF
CUDA_C_32F
CUDA_C_64F

Mixed-precision computation:

A/B	C	computeType
CUDA_R_8I	CUDA_R_32I	CUDA_R_32I
CUDA_R_8I	CUDA_R_32F	CUDA_R_32F
CUDA_R_16F		
CUDA_R_16BF		
CUDA_R_16F	CUDA_R_16F	
CUDA_R_16BF	CUDA_R_16BF	

NOTE: **CUDA_R_16BF/CUDA_C_16BF** data types are supported only with **CUSPARSE_SPMM_COO_ALG4** and **CUSPARSE_SPMM_CSR_ALG2** algorithms

cusparseSpMM supports the following algorithms:

[D]: deprecated

Algorithm	Notes
CUSPARSE_MM_ALG_DEFA [D] CUSPARSE_SPMM_ALG_DE	Default algorithm for any sparse matrix format
CUSPARSE_COOMM_ALG1 [D] CUSPARSE_SPMM_COO_AL	Algorithm 1 for COO sparse matrix format ▸ May provide better performance for small number of nnz ▸ It supports only column-major layout ▸ It supports batched computation ▸ May produce slightly different results during different runs with the same input parameters
CUSPARSE_COOMM_ALG2 [D] CUSPARSE_SPMM_COO_AL	Algorithm 2 for COO sparse matrix format ▸ In general, slower than Algorithm 1 and 2 ▸ It supports only column-major layout ▸ It supports batched computation ▸ It provides deterministc result ▸ It requires additional memory
CUSPARSE_COOMM_ALG3 [D] CUSPARSE_SPMM_COO_AL	Algorithm 3 for COO sparse matrix format ▸ May provide better performance for large number of nnz ▸ It supports only column-major layout

Algorithm	Notes
	▸ May produce slightly different results during different runs with the same input parameters
`CUSPARSE_COOMM_ALG4` [D] `CUSPARSE_SPMM_COO_AL`	Algorithm 4 for COO sparse matrix format ▸ Provide the best performance with row-major layout ▸ It supports batched computation ▸ May produce slightly different results during different runs with the same input parameters
`CUSPARSE_CSRMM_ALG1` [D] `CUSPARSE_SPMM_CSR_AL`	Algorithm 1 for CSR sparse matrix format ▸ It provides deterministc result ▸ It supports only column-major layout
`CUSPARSE_SPMM_CSR_AL`	Algorithm 2 for CSR sparse matrix format ▸ Provide the best performance with row-major layout ▸ It supports batched computation ▸ May produce slightly different results during different runs with the same input parameters

Performance notes:

- Row-major layout provides higher performance than column-major.
- **`CUSPARSE_SPMM_COO_ALG4`** and **`CUSPARSE_SPMM_CSR_ALG2`** should be used with row-major layout, while **`CUSPARSE_SPMM_COO_ALG1`**, **`CUSPARSE_SPMM_COO_ALG2`**, and **`CUSPARSE_SPMM_COO_ALG3`**, and **`CUSPARSE_SPMM_CSR_ALG1`** with column-major layout.
- For **`beta != 1`**, the output matrix is scaled before the actual computation

`cusparseSpMM()` with **`CUSPARSE_SPMM_COO_ALG4`** and **`CUSPARSE_SPMM_CSR_ALG2`** support the following batch modes:

- $C_i = A \cdot B_i$
- $C_i = A_i \cdot B$
- $C_i = A_i \cdot B_i$

The number of batches and their strides can be set by using **`cusparseCooSetStridedBatch`**, **`cusparseCsrSetStridedBatch`**, and **`cusparseDnMatSetStridedBatch`**.

`cusparseSpMM()` has the following properties:

- The routine requires no extra storage for **`CUSPARSE_SPMM_COO_ALG1`**, **`CUSPARSE_SPMM_COO_ALG3`**, **`CUSPARSE_SPMM_COO_ALG4`**, and **`CUSPARSE_SPMM_CSR_ALG1`**
- The routine supports asynchronous execution
- Provides deterministic (bit-wise) results for each run only for **`CUSPARSE_SPMM_COO_ALG2`** and **`CUSPARSE_SPMM_CSR_ALG1`** algorithms, and **`opA == CUSPARSE_OPERATION_NON_TRANSPOSE`**

`cusparseSpMM()` supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseSpMM CSR and cusparseSpMM COO for a code example.

14.6.8. cusparseConstrainedGeMM()

```
cusparseStatus_t
cusparseConstrainedGeMM(cusparseHandle_t           handle,
                        cusparseOperation_t        opA,
                        cusparseOperation_t        opB,
                        const void*                alpha,
                        cusparseDnMatDescr_t matA,
                        cusparseDnMatDescr_t matB,
                        const void*                beta,
                        cusparseSpMatDescr_t       matC,
                        cudaDataType               computeType,
                        void*                      externalBuffer)

cusparseStatus_t
cusparseConstrainedGeMM_bufferSize(cusparseHandle_t     handle,
                                   cusparseOperation_t  opA,
                                   cusparseOperation_t  opB,
                                   const void*          alpha,
                                   cusparseDnMatDescr_t matA,
                                   cusparseDnMatDescr_t matB,
                                   const void*          beta,
                                   cusparseSpMatDescr_t matC,
                                   cudaDataType         computeType,
                                   size_t*              bufferSize)
```

This function performs the multiplication of **`matA`** and **`matB`**, followed by an element-wise multiplication with the sparsity pattern of **`matC`**. Formally, it performs the following operation:

$$\mathbf{C} = \alpha(\mathrm{op}(\mathbf{A}) \cdot \mathrm{op}(\mathbf{B})) \circ \mathrm{spy}(\mathbf{C}) + \beta\mathbf{C}$$

where $\mathrm{op}(\mathbf{A})$ is a dense matrix of size $m \times k$, $\mathrm{op}(\mathbf{B})$ is a dense matrix of size $k \times n$, $\mathbf{C}$ is a sparse matrix of size $m \times n$, α and β are scalars, $\circ$ denotes the Hadamard (entry-wise) matrix product, and $\mathrm{spy}(\mathbf{C})$ is the sparsity pattern matrix of $\mathbf{C}$ defined as:

$$\mathrm{spy}(\mathbf{C})_{ij} = \begin{cases} 0 & \text{if } \mathbf{C}_{ij} = 0 \\ 1 & \text{otherwise} \end{cases}$$

Matrices $\mathrm{op}(\mathbf{A})$ and $\mathrm{op}(\mathbf{B})$ are defined as

$$\mathrm{op}(A) = \begin{cases} A & \text{if opA == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ A^T & \text{if opA == CUSPARSE\_OPERATION\_TRANSPOSE} \\ A^H & \text{if opA == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

$$\mathrm{op}(B) = \begin{cases} B & \text{if opB == CUSPARSE\_OPERATION\_NON\_TRANSPOSE} \\ B^T & \text{if opB == CUSPARSE\_OPERATION\_TRANSPOSE} \\ B^H & \text{if opB == CUSPARSE\_OPERATION\_CONJUGATE\_TRANSPOSE} \end{cases}$$

Param.	Memory	In/out	Meaning
`handle`	HOST	IN	Handle to the cuSPARSE library context
`opA`	HOST	IN	Enumerator specifying the operation $\mathrm{op}(\mathbf{A})$. Has to be `CUSPARSE_OPERATION_NON_TRANSPOSE`
`opB`	HOST	IN	Enumerator specifying the operation $\mathrm{op}(\mathbf{B})$. Has to be `CUSPARSE_OPERATION_NON_TRANSPOSE`
`alpha`	HOST or DEVICE	IN	Scalar α that scales the matrix product
`matA`	HOST	IN	Dense matrix $\mathbf{A}$.
`matB`	HOST	IN	Dense matrix $\mathbf{B}$.
`beta`	HOST or DEVICE	IN	Scalar β that scales the accumulation matrix
`matC`	HOST	IN/OUT	Sparse matrix $\mathbf{C}$.
`computeTyp`	HOST	IN	Enumerator specifying the datatype used to execute the computation
`bufferSize`	HOST	OUT	Size of `externalBuffer` in bytes
`externalBu`	DEVICE	IN	Pointer to a workspace buffer of at least `bufferSize` bytes

Currently, this function only supports `opA == CUSPARSE_OPERATION_NON_TRANSPOSE` and `opB == CUSPARSE_OPERATION_NON_TRANSPOSE`. Attempting to pass a different operator will cause a `CUSPARSE_STATUS_NOT_SUPPORTED` error.

The function has the following limitations:

- Only 32-bit indices `CUSPARSE_INDEX_32I` is supported

The datatypes combinations currrently supported for `cusparseSpMM` are listed below :

Uniform-precision computation:

`A/X/ Y/computeType`
`CUDA_R_16F`
`CUDA_R_32F`
`CUDA_R_64F`
`CUDA_C_16F`
`CUDA_C_32F`
`CUDA_C_64F`

Currently supported sparse matrix formats:

Format	Notes
`CUSPARSE_FORMAT_CSR`	The column indices in each row must be sorted

`cusparseConstrainedGeMM()` has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution

`cusparseConstrainedGeMM()` supports the following optimizations:

- CUDA graph capture
- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

14.6.9. cusparseSpGEMM()

```
cusparseStatus_t CUSPARSEAPI
cusparseSpGEMM_createDescr(cusparseSpGEMMDescr_t* descr);

cusparseStatus_t CUSPARSEAPI
cusparseSpGEMM_destroyDescr(cusparseSpGEMMDescr_t descr);

cusparseStatus_t CUSPARSEAPI
cusparseSpGEMM_workEstimation(cusparseHandle_t      handle,
                              cusparseOperation_t   opA,
                              cusparseOperation_t   opB,
                              const void*           alpha,
                              cusparseSpMatDescr_t  matA,
                              cusparseSpMatDescr_t  matB,
                              const void*           beta,
                              cusparseSpMatDescr_t  matC,
                              cudaDataType          computeType,
                              cusparseSpGEMMAlg_t   alg,
                              cusparseSpGEMMDescr_t spgemmDescr,
                              size_t*               bufferSize1,
                              void*                 externalBuffer1);

cusparseStatus_t CUSPARSEAPI
cusparseSpGEMM_compute(cusparseHandle_t      handle,
                       cusparseOperation_t   opA,
                       cusparseOperation_t   opB,
                       const void*           alpha,
                       cusparseSpMatDescr_t  matA,
                       cusparseSpMatDescr_t  matB,
                       const void*           beta,
                       cusparseSpMatDescr_t  matC,
                       cudaDataType          computeType,
                       cusparseSpGEMMAlg_t   alg,
                       cusparseSpGEMMDescr_t spgemmDescr,
                       void*                 externalBuffer1,
                       size_t*               bufferSize2,
                       void*                 externalBuffer2);

cusparseStatus_t CUSPARSEAPI
cusparseSpGEMM_copy(cusparseHandle_t      handle,
                    cusparseOperation_t   opA,
                    cusparseOperation_t   opB,
                    const void*           alpha,
                    cusparseSpMatDescr_t  matA,
                    cusparseSpMatDescr_t  matB,
                    const void*           beta,
                    cusparseSpMatDescr_t  matC,
                    cudaDataType          computeType,
                    cusparseSpGEMMAlg_t   alg,
                    cusparseSpGEMMDescr_t spgemmDescr,
                    void*                 externalBuffer2);
```

This function performs the multiplication of two sparse matrices **`matA`** and **`matB`**

$$\mathbf{C} = \alpha op(\mathbf{A}) \cdot op(\mathbf{B}) + \beta \mathbf{C}$$

where α and β are scalars.

The example CSR SpGEMM() shows the computation workflow for all steps

The functions **cusparseSpGEMM_workEstimation()** and **cusparseSpGEMM_compute()** are used for both determining the buffer size and performing the actual computation

Param.	Memory	In/out	Meaning
handle	HOST	IN	Handle to the cuSPARSE library context
opA	HOST	IN	Operation **op(A)**
opB	HOST	IN	Operation **op(B)**
alpha	HOST or DEVICE	IN	α scalar used for multiplication
matA	HOST	IN	Sparse matrix **A**
matB	HOST	IN	Dense matrix **B**
beta	HOST or DEVICE	IN	β scalar used for multiplication
matC	HOST	IN/OUT	Dense matrix **C**
computeTyp	HOST	IN	Enumerator specifying the datatype in which the computation is executed
alg	HOST	IN	Enumerator specifying the algorithm for the computation
spgemmDesc	HOST	IN/OUT	Opaque descriptor for storing internal data used across the three steps
bufferSize **bufferSize**	HOST	IN/OUT	Number of bytes of workspace needed by **cusparseSpGEMM_workEstimation** or **cusparseSpGEMM_compute**
externalBu **externalBu**	DEVICE	IN	Pointer to workspace buffer

MEMORY REQUIREMENT: the first invocation of **cusparseSpGEMM_compute** provides an *upper bound* of the memory required for the computation that is generally several times larger of the actual memory used. The user can provide an arbitrary buffer size **bufferSize2** in the second invocation. If it is not sufficient, the routine will returns **CUSPARSE_STATUS_INSUFFICIENT_RESOURCES** status.

Currently, the function has the following limitations:

- Only 32-bit indices **CUSPARSE_INDEX_32I** is supported
- Only CSR format **CUSPARSE_FORMAT_CSR** is supported
- Only **opA**, **opB** equal to **CUSPARSE_OPERATION_NON_TRANSPOSE** are supported

The datatypes combinations currrently supported for **cusparseSpGEMM** are listed below :

Uniform-precision computation:

A/B/ C/computeType
CUDA_R_16F
CUDA_R_16BF
CUDA_R_32F

A/B/ C/computeType
CUDA_R_64F
CUDA_C_16F
CUDA_C_16BF
CUDA_C_32F
CUDA_C_64F

cusparseSpGEMM routine runs for the following algorithm:

Algorithm	Notes
CUSPARSE_SPGEMM_DEFA	Default algorithm. Provides deterministic (bit-wise) results for each run

cusparseSpGEMM() has the following properties:

- The routine requires no extra storage
- The routine supports asynchronous execution
- The routine supports does ***not*** support CUDA graph capture

cusparseSpGEMM() supports the following optimizations:

- Hardware Memory Compression

See `cusparseStatus_t` for the description of the return status

Please visit cuSPARSE Library Samples - cusparseSpGEMM for a code example.

Chapter 15. APPENDIX B: CUSPARSE FORTRAN BINDINGS

The cuSPARSE library is implemented using the C-based CUDA toolchain, and it thus provides a C-style API that makes interfacing to applications written in C or C++ trivial. There are also many applications implemented in Fortran that would benefit from using cuSPARSE, and therefore a cuSPARSE Fortran interface has been developed.

Unfortunately, Fortran-to-C calling conventions are not standardized and differ by platform and toolchain. In particular, differences may exist in the following areas:

Symbol names (capitalization, name decoration)

Argument passing (by value or reference)

Passing of pointer arguments (size of the pointer)

To provide maximum flexibility in addressing those differences, the cuSPARSE Fortran interface is provided in the form of wrapper functions, which are written in C and are located in the file **`cusparse_fortran.c`**. This file also contains a few additional wrapper functions (for **`cudaMalloc()`**, **`cudaMemset`**, and so on) that can be used to allocate memory on the GPU.

The cuSPARSE Fortran wrapper code is provided as an example only and needs to be compiled into an application for it to call the cuSPARSE API functions. Providing this source code allows users to make any changes necessary for a particular platform and toolchain.

The cuSPARSE Fortran wrapper code has been used to demonstrate interoperability with the compilers g95 0.91 (on 32-bit and 64-bit Linux) and g95 0.92 (on 32-bit and 64-bit Mac OS X). In order to use other compilers, users have to make any changes to the wrapper code that may be required.

The direct wrappers, intended for production code, substitute device pointers for vector and matrix arguments in all cuSPARSE functions. To use these interfaces, existing applications need to be modified slightly to allocate and deallocate data structures in GPU memory space (using **`CUDA_MALLOC()`** and **`CUDA_FREE()`**) and to copy data between GPU and CPU memory spaces (using the **`CUDA_MEMCPY()`** routines). The sample wrappers provided in **`cusparse_fortran.c`** map device pointers to the OSdependent type **size_t**, which is 32 bits wide on 32-bit platforms and 64 bits wide on a 64-bit platforms.

One approach to dealing with index arithmetic on device pointers in Fortran code is to use C-style macros and to use the C preprocessor to expand them. On Linux and Mac OS X, preprocessing can be done by using the option **'-cpp'** with g95 or gfortran. The function **GET_SHIFTED_ADDRESS()**, provided with the cuSPARSE Fortran wrappers, can also be used, as shown in example B.

Example B shows the the C++ of example A implemented in Fortran 77 on the host. This example should be compiled with **ARCH_64** defined as 1 on a 64-bit OS system and as undefined on a 32-bit OS system. For example, on g95 or gfortran, it can be done directly on the command line using the option **-cpp -DARCH_64=1**.

15.1. Fortran Application

```
c     #define ARCH_64 0
c     #define ARCH_64 1

      program cusparse_fortran_example
      implicit none
      integer cuda_malloc
      external cuda_free
      integer cuda_memcpy_c2fort_int
      integer cuda_memcpy_c2fort_real
      integer cuda_memcpy_fort2c_int
      integer cuda_memcpy_fort2c_real
      integer cuda_memset
      integer cusparse_create
      external cusparse_destroy
      integer cusparse_get_version
      integer cusparse_create_mat_descr
      external cusparse_destroy_mat_descr
      integer cusparse_set_mat_type
      integer cusparse_get_mat_type
      integer cusparse_get_mat_fill_mode
      integer cusparse_get_mat_diag_type
      integer cusparse_set_mat_index_base
      integer cusparse_get_mat_index_base
      integer cusparse_xcoo2csr
      integer cusparse_dsctr
      integer cusparse_dcsrmv
      integer cusparse_dcsrmm
      external get_shifted_address
#if ARCH_64
      integer*8 handle
      integer*8 descrA
      integer*8 cooRowIndex
      integer*8 cooColIndex
      integer*8 cooVal
      integer*8 xInd
      integer*8 xVal
      integer*8 y
      integer*8 z
      integer*8 csrRowPtr
      integer*8 ynp1
#else
      integer*4 handle
      integer*4 descrA
      integer*4 cooRowIndex
      integer*4 cooColIndex
      integer*4 cooVal
      integer*4 xInd
      integer*4 xVal
      integer*4 y
      integer*4 z
      integer*4 csrRowPtr
      integer*4 ynp1
#endif
      integer status
      integer cudaStat1,cudaStat2,cudaStat3
      integer cudaStat4,cudaStat5,cudaStat6
      integer n, nnz, nnz_vector
      parameter (n=4, nnz=9, nnz_vector=3)
      integer cooRowIndexHostPtr(nnz)
      integer cooColIndexHostPtr(nnz)
      real*8  cooValHostPtr(nnz)
      integer xIndHostPtr(nnz_vector)
      real*8  xValHostPtr(nnz_vector)
      real*8  yHostPtr(2*n)
      real*8  zHostPtr(2*(n+1))
      integer i, j
      integer version, mtype, fmode, dtype, ibase
      real*8  dzero,dtwo,dthree,dfive
      real*8  epsilon


      write(*,*) "testing fortran example"
```

Chapter 16. APPENDIX B: EXAMPLES OF SORTING

16.1. COO Sort

This chapter provides a simple example in the C programming language of sorting of COO format.

A is a 3x3 sparse matrix,

$$A = \begin{pmatrix} 1.0 & 2.0 & 0.0 \\ 0.0 & 5.0 & 0.0 \\ 0.0 & 8.0 & 0.0 \end{pmatrix}$$

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include coosort.cpp
 *   g++ -o coosort.cpp coosort.o -L/usr/local/cuda/lib64 -lcusparse -lcudart
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;

    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    cudaError_t cudaStat6 = cudaSuccess;

/*
 * A is a 3x3 sparse matrix
 *     | 1 2 0 |
 * A = | 0 5 0 |
 *     | 0 8 0 |
 */
    const int m = 3;
    const int n = 3;
    const int nnz = 4;

#if 0
/* index starts at 0 */
    int h_cooRows[nnz] = {2, 1, 0, 0 };
    int h_cooCols[nnz] = {1, 1, 0, 1 };
#else
/* index starts at -2 */
    int h_cooRows[nnz] = {0, -1, -2, -2 };
    int h_cooCols[nnz] = {-1, -1, -2, -1 };
#endif
    double h_cooVals[nnz] = {8.0, 5.0, 1.0, 2.0 };
    int h_P[nnz];

    int *d_cooRows = NULL;
    int *d_cooCols = NULL;
    int *d_P       = NULL;
    double *d_cooVals = NULL;
    double *d_cooVals_sorted = NULL;
    size_t pBufferSizeInBytes = 0;
    void *pBuffer = NULL;

    printf("m = %d, n = %d, nnz=%d \n", m, n, nnz );
```

```
/* step 1: create cusparse handle, bind a stream */
    cudaStat1 = cudaStreamCreateWithFlags(&stream, cudaStreamNonBlocking);
    assert(cudaSuccess == cudaStat1);

    status = cusparseCreate(&handle);
    assert(CUSPARSE_STATUS_SUCCESS == status);

    status = cusparseSetStream(handle, stream);
    assert(CUSPARSE_STATUS_SUCCESS == status);

/* step 2: allocate buffer */
```

Chapter 17. APPENDIX C: EXAMPLES OF PRUNE

17.1. Prune Dense to Sparse

This section provides a simple example in the C programming language of pruning a dense matrix to a sparse matrix of CSR format.

A is a 4x4 dense matrix,

$$A = \begin{pmatrix} 1.0 & 0.0 & 2.0 & -3.0 \\ 0.0 & 4.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 6.0 & 7.0 \\ 0.0 & 8.0 & 0.0 & 9.0 \end{pmatrix}$$

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunedense_example.cpp
 *   g++ -o prunedense_example.cpp prunedense_example.o -L/usr/local/cuda/lib64
 -lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printMatrix(int m, int n, const float*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            float Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d\n", name, m, n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrC = NULL;
```

```
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int lda = m;
/*
 *      |    1     0     2     -3   |
 *      |    0     4     0     0    |
 *  A = |    5     0     6     7    |
```

17.2. Prune Sparse to Sparse

This section provides a simple example in the C programming language of pruning a sparse matrix to a sparse matrix of CSR format.

A is a 4x4 sparse matrix,

$$A = \begin{pmatrix} 1.0 & 0.0 & 2.0 & -3.0 \\ 0.0 & 4.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 6.0 & 7.0 \\ 0.0 & 8.0 & 0.0 & 9.0 \end{pmatrix}$$

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunecsr_example.cpp
 *   g++ -o prunecsr_example.cpp prunecsr_example.o -L/usr/local/cuda/lib64 -
lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m,
 n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseMatDescr_t descrC = NULL;

    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int nnzA = 9;
/*
 *      |    1     0     2     -3   |
 *      |    0     4     0     0    |
 *   A = |    5     0     6     7    |
 *      |    0     8     0     9    |
 *
 */
```

```
    const int csrRowPtrA[m+1] = { 1, 4, 5, 8, 10};
    const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
    const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};

    int* csrRowPtrC = NULL;
    int* csrColIndC = NULL;
    float* csrValC  = NULL;
```

17.3. Prune Dense to Sparse by Percentage

This section provides a simple example in the C programming language of pruning a dense matrix to a sparse matrix by percentage.

A is a 4x4 dense matrix,

$$A = \begin{pmatrix} 1.0 & 0.0 & 2.0 & -3.0 \\ 0.0 & 4.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 6.0 & 7.0 \\ 0.0 & 8.0 & 0.0 & 9.0 \end{pmatrix}$$

The percentage is 50, which means to prune 50 percent of the dense matrix. The matrix has 16 elements, so 8 out of 16 must be pruned out. Therefore 7 zeros are pruned out, and value 1.0 is also out because it is the smallest among 9 nonzero elements.

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunedense2csrbyP.cpp
 *   g++ -o prunedense2csrbyP.cpp prunedense2csrbyP.o -L/usr/local/cuda/lib64 -
lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printMatrix(int m, int n, const float*A, int lda, const char* name)
{
    for(int row = 0 ; row < m ; row++){
        for(int col = 0 ; col < n ; col++){
            float Areg = A[row + col*lda];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m,
 n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrC = NULL;
    pruneInfo_t info = NULL;

    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    cudaError_t cudaStat2 = cudaSuccess;
    cudaError_t cudaStat3 = cudaSuccess;
    cudaError_t cudaStat4 = cudaSuccess;
    cudaError_t cudaStat5 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int lda = m;
```



```
/*
 *      |    1     0     2     -3   |
 *      |    0     4     0      0   |
```

17.4. Prune Sparse to Sparse by Percentage

This section provides a simple example in the C programming language of pruning a sparse matrix to a sparse matrix by percentage.

A is a 4x4 sparse matrix,

$$A = \begin{pmatrix} 1.0 & 0.0 & 2.0 & -3.0 \\ 0.0 & 4.0 & 0.0 & 0.0 \\ 5.0 & 0.0 & 6.0 & 7.0 \\ 0.0 & 8.0 & 0.0 & 9.0 \end{pmatrix}$$

The percentage is 20, which means to prune 20 percent of the nonzeros. The sparse matrix has 9 nonzero elements, so 1.4 elements must be pruned out. The function removes 1.0 and 2.0 which are first two smallest numbers of nonzeros.

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include prunecsr2csrByP.cpp
 *   g++ -o prunecsr2csrByP.cpp prunecsr2csrByP.o -L/usr/local/cuda/lib64 -
lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

void printCsr(
    int m,
    int n,
    int nnz,
    const cusparseMatDescr_t descrA,
    const float *csrValA,
    const int *csrRowPtrA,
    const int *csrColIndA,
    const char* name)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;

    printf("matrix %s is %d-by-%d, nnz=%d, base=%d, output base-1\n", name, m,
 n, nnz, base);
    for(int row = 0 ; row < m ; row++){
        const int start = csrRowPtrA[row  ] - base;
        const int end   = csrRowPtrA[row+1] - base;
        for(int colidx = start ; colidx < end ; colidx++){
            const int col = csrColIndA[colidx] - base;
            const float Areg = csrValA[colidx];
            printf("%s(%d,%d) = %f\n", name, row+1, col+1, Areg);
        }
    }
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    cusparseMatDescr_t descrC = NULL;
    pruneInfo_t info = NULL;

    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int m = 4;
    const int n = 4;
    const int nnzA = 9;
/*
 *      |    1     0     2     -3  |
 *      |    0     4     0     0   |
 *  A = |    5     0     6     7   |
 *      |    0     8     0     9   |
 *
 */
```

```
    const int csrRowPtrA[m+1] = { 1, 4, 5, 8, 10};
    const int csrColIndA[nnzA] = { 1, 3, 4, 2, 1, 3, 4, 2, 4};
    const float csrValA[nnzA] = {1, 2, -3, 4, 5, 6, 7, 8, 9};

    int* csrRowPtrC = NULL;
    int* csrColIndC = NULL;
    float* csrValC  = NULL;
```

Chapter 18. APPENDIX D: EXAMPLES OF GPSV

18.1. Batched Penta-diagonal Solver

This section provides a simple example in the C programming language of gpsvInterleavedBatch .

The example solves two penta-diagonal systems and assumes data layout is NOT interleaved format. Before calling gpsvInterleavedBatch, cublasXgeam is used to transform the data layout, from aggregate format to interleaved format. If the user can prepare interleaved format, no need to transpose the data.

```
*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include gpsv.cpp
 *   g++ -o gpsv gpsv.o -L/usr/local/cuda/lib64 -lcusparse -lcublas -lcudart
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>
#include <cublas_v2.h>

/*
 * compute | b - A*x|_inf
 */
void residaul_eval(
    int n,
    const float *ds,
    const float *dl,
    const float *d,
    const float *du,
    const float *dw,
    const float *b,
    const float *x,
    float *r_nrminf_ptr)
{
    float r_nrminf = 0;
    for(int i = 0 ; i < n ; i++){
        float dot = 0;
        if (i > 1 ){
            dot += ds[i]*x[i-2];
        }
        if (i > 0 ){
            dot += dl[i]*x[i-1];
        }
        dot += d[i]*x[i];
        if (i < (n-1) ){
            dot += du[i]*x[i+1];
        }
        if (i < (n-2) ){
            dot += dw[i]*x[i+2];
        }
        float ri = b[i] - dot;
        r_nrminf = (r_nrminf > fabs(ri))? r_nrminf : fabs(ri);
    }

    *r_nrminf_ptr = r_nrminf;
}

int main(int argc, char*argv[])
{
    cusparseHandle_t cusparseH = NULL;
    cublasHandle_t cublasH = NULL;
    cudaStream_t stream = NULL;
```

```
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cublasStatus_t cublasStat = CUBLAS_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;

    const int n = 4;
    const int batchSize = 2;

/*
 *      |  1    8   13    0  |       | 1 |       | -0.0592 |
 *   A1 =|  5    2    9   14  |, b1 = | 2 |, x1 = |  0.3428 |
 *      | 11    6    3   10  |       | 3 |       | -0.1295 |
 *      |  0   12    7    4  |       | 4 |       |  0.1982 |
```

Chapter 19. APPENDIX E: EXAMPLES OF CSRSM2

19.1. Forward Triangular Solver

This section provides a simple example in the C programming language of csrsm2.

The example solves a lower triangular system with 2 right hand side vectors.

```
/*
 * How to compile (assume cuda is installed at /usr/local/cuda/)
 *   nvcc -c -I/usr/local/cuda/include csrms2.cpp
 *   g++ -o csrm2 csrsm2.o -L/usr/local/cuda/lib64 -lcusparse -lcudart
 */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <cuda_runtime.h>
#include <cusparse.h>

/* compute | b - A*x|_inf */
void residaul_eval(
    int n,
    const cusparseMatDescr_t descrA,
    const float *csrVal,
    const int *csrRowPtr,
    const int *csrColInd,
    const float *b,
    const float *x,
    float *r_nrminf_ptr)
{
    const int base = (cusparseGetMatIndexBase(descrA) !=
 CUSPARSE_INDEX_BASE_ONE)? 0:1 ;
    const int lower = (CUSPARSE_FILL_MODE_LOWER ==
 cusparseGetMatFillMode(descrA))? 1:0;
    const int unit  = (CUSPARSE_DIAG_TYPE_UNIT ==
 cusparseGetMatDiagType(descrA))? 1:0;

    float r_nrminf = 0;
    for(int row = 0 ; row < n ; row++){
        const int start = csrRowPtr[row]   - base;
        const int end   = csrRowPtr[row+1] - base;
        float dot = 0;
        for(int colidx = start ; colidx < end; colidx++){
            const int col = csrColInd[colidx] - base;
            float Aij = csrVal[colidx];
            float xj  = x[col];
            if ( (row == col) && unit ){
                Aij = 1.0;
            }
            int valid = (row >= col) && lower ||
                        (row <= col) && !lower ;
            if ( valid ){
                dot += Aij*xj;
            }
        }
        float ri = b[row] - dot;
        r_nrminf = (r_nrminf > fabs(ri))? r_nrminf : fabs(ri);
    }
    *r_nrminf_ptr = r_nrminf;
}

int main(int argc, char*argv[])
{
    cusparseHandle_t handle = NULL;
    cudaStream_t stream = NULL;
    cusparseMatDescr_t descrA = NULL;
    csrsm2Info_t info = NULL;
```

```
    cusparseStatus_t status = CUSPARSE_STATUS_SUCCESS;
    cudaError_t cudaStat1 = cudaSuccess;
    const int nrhs = 2;
    const int n = 4;
    const int nnzA = 9;
    const cusparseSolvePolicy_t policy = CUSPARSE_SOLVE_POLICY_NO_LEVEL;
    const float h_one = 1.0;
/*
 *      |    1     0     2     -3  |
 *      |    0     4     0     0   |
```

Chapter 20. APPENDIX F: ACKNOWLEDGEMENTS

NVIDIA would like to thank the following individuals and institutions for their contributions:

- The cusparse<t>gtsv implementation is derived from a version developed by Li-Wen Chang from the University of Illinois.
- the cusparse<t>gtsvInterleavedBatch adopts cuThomasBatch developed by Pedro Valero-Lara and Ivan Martínez-Pérez from Barcelona Supercomputing Center and BSC/UPC NVIDIA GPU Center of Excellence.

Chapter 21. BIBLIOGRAPHY

[1] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processors", Supercomputing, 2009.

[2] R. Grimes, D. Kincaid, and D. Young, "ITPACK 2.0 User's Guide", Technical Report CNA-150, Center for Numerical Analysis, University of Texas, 1979.

[3] M. Naumov, "Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cuSPARSE and cuBLAS", Technical Report and White Paper, 2011.

[4] Pedro Valero-Lara, Ivan Martínez-Pérez, Raül Sirvent, Xavier Martorell, and Antonio J. Peña. NVIDIA GPUs Scalability to Solve Multiple (Batch) Tridiagonal Systems. Implementation of cuThomasBatch. In Parallel Processing and Applied Mathematics - 12th International Conference (PPAM), 2017.